Logical Methods in Computer Science
Volume 15, Issue 1, 2019, pp. 34:1–34:37
https://lmcs.episciences.org/

Submitted Jan. 24, 2018
Published Mar. 29, 2019

COMPUTER-ASSISTED PROVING OF COMBINATORIAL CONJECTURES OVER FINITE DOMAINS: A CASE STUDY OF A CHESS CONJECTURE

PREDRAG JANIČIĆ [a], FILIP MARIĆ [a], AND MARKO MALIKOVIĆ [b]

[a] Faculty of Mathematics, University of Belgrade, Serbia
e-mail address: {janicic,filip}@matf.bg.ac.rs

[b] Faculty of Humanities and Social Sciences, University of Rijeka, Croatia
e-mail address: marko@ffri.hr

Abstract. There are several approaches for using computers in deriving mathematical proofs. For their illustration, we provide an in-depth study of using computer support for proving one complex combinatorial conjecture – correctness of a strategy for the chess KRK endgame. The final, machine verifiable result presented in this paper is that there is a winning strategy for white in the KRK endgame generalized to $n \times n$ board (for natural n greater than 3). We demonstrate that different approaches for computer-based theorem proving work best together and in synergy and that the technology currently available is powerful enough for providing significant help to humans deriving some complex proofs.

1. Introduction

Over the last several decades, automated and interactive theorem provers have made huge advances which changed the mathematical landscape significantly. Theorem provers are already widely used in many areas of mathematics and computer science, and there are already proofs of many extremely complex theorems developed within proof assistants and with many lemmas proved or checked automatically [25, 31, 33]. We believe there are changes still to come, changes that would make new common mathematical practices and proving process will be more widely supported by tools that automatically and reliably prove some conjectures and even discover new theorems. Generally, proving mathematical conjectures can be assisted by computers in several forms:

- for exhaustive analysis (e.g., for checking of all possible cases);
- for automated proving of relevant statements (e.g., by generic automated provers or by solvers for specific theories);
- for interactive theorem proving (e.g., for proving correctness of exhaustive analysis algorithms or for direct proving of relevant statements).

Key words and phrases: Chess, chess endgame, strategy, theorem proving, proof assistants, SAT, SMT, constraint programming .

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.23638/LMCS-15(1:34)2019

Each of the above forms of support provides different sorts of arguments, each has its limitations, and its strengths and weaknesses. In this paper, we advocate that it is their synergy that provides a way for proving some complex combinatorial conjectures. Namely, in every proving process, a human mathematician experiments, analyses special cases, tries to discover or prove simpler conjectures, etc. However, all these are typically hidden in the final product and published mathematics is typically the art of polished proofs, rarely the art of how to reach them. Computer support can be crucial in the demanding process of seeking and proving new mathematical truths. However, for each computer-supported proving approach, one has to consider the following key questions:

- What have we *really* proved?
- Is our proof a real mathematical proof, or just a supporting argument?
- How reliable is our proof?
- What was the level of automation and the level of human effort required to make the proof?

As a case study, we use a conjecture from one of favourite domains for many AI approaches – chess. We consider a conjecture that states *correctness of a strategy for one chess endgame*. Endgame strategies provide concise, understandable, and intuitive instructions for the player and correctness means that the strategy always leads to the best possible outcome (under any play by the opponent). We show that both chess strategies and proofs of their correctness can be rigorously formalized, i.e., expressed in pure mathematical terms. Although a strategy for an endgame such as KRK (white king and white rook against black king) is simple, its formalization has a number of details and it is not easy to prove its correctness. One of our goals and contributions is modelling the chess rules and chess endgames so that the correctness proofs can be made as automated, efficient and reliable as possible.

Through this complex exercise, we will show that a real-world process of proving conjectures such as correctness of a chess strategy can be naturally based on a synergy between different computer-supported approaches and combines experimentation, testing special cases, checking properties, exploring counterexamples for some conjectures, and, at the end, proving conjectures within a proof assistant, using as much automation as possible. Correctness of the strategy for the KRK endgame is just an example, and the main purpose of this work is to illustrate a methodology that can be used in proving some complex combinatorial conjectures in an efficient and a highly reliable way. In our previous work, we proved the correctness of a very similar KRK strategy within a constraint solving system URSA [30] and a proof assistant Isabelle/HOL[35]. In this paper, we give a unifying perspective of proving conjectures like that one using three kinds of systems – a general purpose programming language, a constraint solver, and a proof assistant, and we further improve earlier proofs in terms of efficiency, reliability, understandability and generality.

Overview of the paper. In Section 2 we give some background on automated theorem proving, SAT and SMT, constraint programming systems and URSA, interactive theorem proving and Isabelle/HOL. In Section 3 we discuss formal representation of chess and chess endgames in various logic and languages. In Section 4 we discuss different methods for reasoning about chess endgames and proving their correctness. In Section 5 we discuss different methods to make our proofs faster, and our conjectures higher-level and more general. In Section 6 we discuss some related work and in Section 7 we draw final conclusions.

2. Background

2.1. **Automated and Interactive Theorem Proving.**

Automated theorem proving, SAT and SMT. Modern automated theorem provers based on uniform procedures, such as the resolution method, and also specific solvers, such as SAT and SMT solvers, can decide validity of huge formula coming a wide spectrum of areas including software and hardware verification, model checking, termination analysis, planning, scheduling, cryptanalysis, etc. [7]. One of the most widely used SMT theories is linear arithmetic, a decidable fragment of arithmetic (over integers – LIA, or reals – LRA) that uses only addition – multiplication is only allowed by a constant number, and nx is just a shorthand for $x + x + \ldots + x$ where x occurs n times. Linear arithmetic is rather simple, but expressible enough to be widely used in applications in computer science [7]. There are several decision procedures for variants of linear arithmetic and they are widely available through modern SMT solvers [20].

Constraint programming systems and URSA. Constraint programming systems allow specifying problems and searching for models that meet given conditions, by using various approaches (e.g., constraint logic programming over finite domains, answer set programming, disjunctive logic programming). Some constraint systems, such as URSA [30], are based on reduction to propositional satisfiability problem (SAT). In URSA, the problem is specified in a language which is imperative and similar to C, but at the same time, is declarative, as the user does not have to provide a solving mechanism for the given problem. URSA allows two types of variables: (unsigned) numerical (with names beginning with `n`, e.g., `nX`) and Boolean (with names beginning with `b`, e.g., `bX`), with a wide range of C-like operators (arithmetic, relational, logical, and bitwise). Variables can have concrete (ground) or symbolic values (in which case, they are represented by vectors of propositional formulae). There is support for procedures and there are control-flow structures (in the style of C). Loops must be with known bounds and there is no `if-else` statement, but only `ite` expression (corresponding to `?:` in C). An URSA specification is symbolically executed and the given constraint corresponds to one propositional formula. It is then transformed into CNF and passed to one of the underlying SAT solvers. If this formula is satisfiable, the system can return all its models.

Interactive theorem proving and Isabelle/HOL. Interactive theorem provers or *proof assistants* are systems used to check proofs constructed by the user, by verifying each proof step with respect to the given underlying logic [48]. Proofs written within proof assistants are typically much longer than traditional, pen-and-paper proofs [5] and are considered to be very reliable [4]. Modern proof assistants support a high-level of automation and significant parts of proofs can be constructed automatically. Some proof assistants are also connected to powerful external automated theorem provers and SMT solvers, and thanks to that are now capable of proving very complex combinatorial conjectures.

Isabelle [38] is a generic proof assistant, but its most developed application is higher order logic (Isabelle/HOL). Formalizations of mathematical theories are made by defining new notions (types, constants, functions, etc.), and proving statements about them (lemmas, theorems, etc.). This is often done using the declarative proof language Isabelle/Isar [47]. Isabelle/HOL incorporates several automated provers (e.g., classical reasoner, simplifier)

and it has been connected to SMT solvers [9], enabling users to employ SMT solvers to discharge some goals that arise in interactive theorem proving.

2.2. **Chess Endgame Strategies.** Techniques used by computer programs for chess in midgames (minimax-style algorithms) are often not appropriate for endgames and then other techniques have to be used. One such technique is based on lookup tables (i.e., endgame databases) with pre-calculated optimal moves for each legal position. However, such tables for endgames with more chess pieces require a lot of memory and, in addition, they are completely useless for human players.[1]. One alternative to huge lookup tables, usable both to human and computer players, are *endgame strategies*. Endgame strategies are algorithms that provide concise, understandable, and intuitive instructions for the player. Endgame strategies do not need to ensure optimal moves (e.g., shortest path winning moves), but must ensure correctness – i.e., if a player A follows the strategy, he/she should always reach the best possible outcome. The main focus of our work is formal analysis of combinatorial algorithms, so we will consider only endgame strategies (and not endgame databases).

One of the simplest chess endgames is the KRK endgame. There are several winning strategies for white for this endgame. Some of these were designed by humans, while some are generated semi-automatically or automatically, using endgame databases, certain sets of human advices, and approaches such as inductive logic programming, genetic programming, neural networks, machine learning, etc. However, only a few of them are really human-understandable. Some strategies for white were proposed by Zuidema [49] (a strategy based on a high-level advice instead of search), Bratko [13, 15, 14] (an advice-based strategy consisting of several sorts of strategic moves), Seidel [42] (a strategy using the ring structure of the chessboard), Morales [36, 37] (short strategies produced by inductive logic programming assisted by a human), etc.

2.2.1. *Bratko-Style Strategy for White for the KRK Endgame.* In the rest of the paper, we will consider one strategy for white for KRK: it is a variation of Bratko's strategy and slightly modified with respect to the version published earlier [32].

We assume standard chess notions such as legal moves, mate, stalemate, etc. (as defined in the FIDE Handbook [22]). *Legal KRK positions* contain three pieces: the white king (WK), the white rook (WR), and the black king (BK). On the 8×8 board, there are 399 112 such positions, while the strategy is applied only to 175 168 of them – those with white on turn.

Auxiliary Notions. In the following text, we assume that files (columns) and ranks (rows) of the chessboard are associated with the numbers $0, 1, \ldots, 7$, and the squares are represented by (x, y) pairs of natural numbers between 0 and 7. For formulating the strategy, we use several auxiliary notions (standard notions or notions introduced by Bratko):
Manhattan distance: For two squares of the chessboard (x_1, y_1) and (x_2, y_2), the Manhattan distance equals $|x_1 - x_2| + |y_1 - y_2|$.

[1]The Lomonosov Endgame Tablebases that contain optimal play for all endgames with seven or less pieces, generated by Zakharov and Makhnichev (`http://tb7.chessok.com/`) have around 140 Terabytes. It was shown that there is a position with seven pieces such that black can be mated in 545 moves but not in less moves, if she/he plays optimally

Chebyshev distance: For two squares of the chessboard (x_1, y_1) and (x_2, y_2), the Chebyshev distance is the minimal number of moves a king requires to move between them, i.e., $\max(|x_1 - x_2|, |y_1 - y_2|)$.

Room: Following the strategy, white tries to squeeze the rectangular space available to black king – that space is called the *room* (Figure 1) and is measured by its half-perimeter. When the black king and the white rook are in line, the black king is not confined (not restricted to a rectangular area), and the room takes the value 15 (whenever the black king is confined, the half-perimeter of the guarded space is at most 14)[2]. Therefore, if the rook is on the square (wr_x, wr_y) and the black king on the square (bk_x, bk_y), then room equals:

$$\begin{cases} 15, & \text{if } wr_x = bk_x \text{ or } wr_y = bk_y \\ x + y, & \text{otherwise} \end{cases}$$

where x and y are lengths of sides of the guarded space (it holds that $x = wr_x$ if $wr_x > bk_x$ and $x = 7 - wr_x$ if $wr_x < bk_x$, and analogously for y).

Critical square: The *critical square* is the square adjacent to the square of the rook in the direction of the black king; if the rook and the black king are in the same column or the same row, then the critical square is between them, otherwise, it is diagonal to the square of the rook (Figure 1). More precisely, if the rook is on (wr_x, wr_y) and the black king on (bk_x, bk_y), then the coordinates (x, y) of the critical square are given as follows:

$$x = \begin{cases} wr_x, & \text{if } wr_x = bk_x \\ wr_x - 1, & \text{if } wr_x > bk_x \\ wr_x + 1, & \text{if } wr_x < bk_x \end{cases} \qquad y = \begin{cases} wr_y, & \text{if } wr_y = bk_y \\ wr_y - 1, & \text{if } wr_y > bk_y \\ wr_y + 1, & \text{if } wr_y < bk_y \end{cases}$$

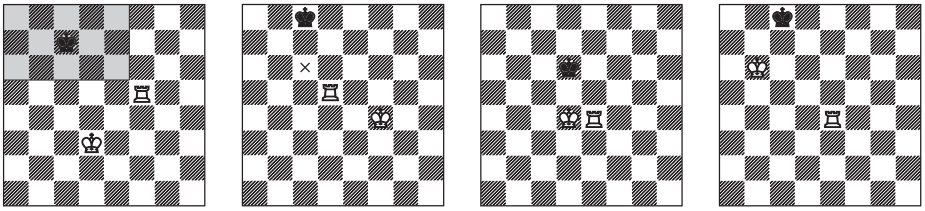

Figure 1: From left to right: illustration of the notion of *room*, of the notion of *critical square*, of the notion of *L-pattern*, and of a position in which the *ReadyToMate* move is possible

Rook exposed: The rook is *exposed* if white king cannot reach it fast enough to protect it, i.e., if white (black) is on turn and the Chebyshev distance between the rook and the white king is greater by at least 2 (by at least 1) than the Chebyshev distance between the rook and the black king.

Rook divides: The white rook *divides* two kings if its x coordinate is (strictly) between x coordinates of the two kings, or if its y coordinate is (strictly) between y coordinates of the two kings (or both).

[2]On the $n \times n$ board, the half-perimeter is at most $2n - 2$, and when the black king is not confined, the room is $2n - 1$.

L-pattern: Three KRK pieces form an *L-pattern* if the kings are in the same row (column), at the distance 2, and if the rook and the white king are in the same column (row) and at the distance 1 (Figure 1).
Kings on a same edge: The two kings are on a same edge if their files or ranks are both equal to 0 or 7.
Towards black king's edge move: If the black king is on an edge, then the white king moves towards that edge.

Basic Strategy. The strategy can be defined as follows:

1. *ImmediateMate*: If there is a mating move, play it;
2. *ReadyToMate*: If the above is not possible and there is a move that leads to mate in the next move, play it (Figure 1).
3. *Squeeze*: If none of the above is possible, make a move (by the rook) that reduces the room and in the reached position it holds that: (*i*) the rook is not exposed, (*ii*) the rook divides the kings and (*iii*) it is not stalemate.
4. *Approach*: If none of the above is possible, then approach the critical square, i.e., move the king so the Manhattan distance between the king and the critical square is decreased; in the reached position, the following has to hold: (*i*) the rook is not exposed, (*ii*) the rook divides the kings or there is a L-pattern, (*iii*) if the room is less than or equal to 3, then the following holds: the white king is not on an edge and if its Chebyshev distance from the rook is 1, then it does not make a move towards the black king's edge, and (*iv*) it is not stalemate. Play the approach move in a non-diagonal direction (*ApproachNonDiag*) only if no diagonal approach move (*ApproachDiag*) is possible.
5. *KeepRoom*: If none of the above is possible, then keep the room, i.e., move the king if that does not increase the Chebyshev distance from the rook; in the reached position, the following has to hold: (*i*) the rook is not exposed and divides the kings, and (*ii*) if the room is less than or equal to 3, then the following holds: the white king is not on an edge and if its Chebyshev distance from the rook is 1, then it does not make a move towards the black king's edge, and (*iii*) it is not stalemate. Play the keep room move in a non-diagonal direction (*KeepRoomNonDiag*) only if no diagonal keep room move (*KeepRoomDiag*) is possible.
6. *RookHome*: If none of the above is possible, then move the rook to be horizontally or vertically adjacent to the white king; in the reached position the following has to hold: (*i*) the rook is adjacent to the black king only if it is guarded by the white king and (*ii*) it is not stalemate.
7. *RookSafe*: If none of the above is possible, then move the rook to some edge (other than the edge it is possibly is on); in the reached position the following has to hold: (*i*) either both kings are next to the rook or the Chebyshev distance between the rook and the black king is greater[3] than 2, and (*ii*) it is not stalemate.

On the 8×8 board, the above steps are used in the following number of positions (in total 175168): 1512 (*ImmediateMate*), 4676 (*ReadyToMate*), 116504 (*Squeeze*), 12160+4020 (*ApproachNonDiag* + *ApproachDiag*), 3160+184 (*KeepRoomNonDiag* + *KeepRoomDiag*), 32520 (*RookHome*), 432 (*RookSafe*). Note that some kinds of strategy moves can be played in different ways. For example, when the *Squeeze* move is applicable, there are often several

[3]If the distance would be equal to 2, then the black king could approach the rook, so the rook would have to make the rook safe move again and again.

possibilities to play it. Although it is not necessary for correctness (as we will show), for efficiency reasons (i.e., for reaching mate faster) *Squeeze* should be *maximal* – it should be played so that the black king is confined to the smallest possible room. Also, if there are more *RookHome* moves applicable, the one with the smallest Manhattan distance between the rook and the black king should be chosen. For other moves for which there are more options, it does not matter which of them will be selected.

For the sake of formal analysis, the above strategy differ to some extent from Bratko's strategy, but still keeps its spirit [32].

3. Problem Representation

The FIDE Handbook [22] is the authoritative account of the laws of chess, but it specifies chess only informally and is suitable only for informal reasoning. For more rigorous reasoning, the chess notions have to be specified formally, in some strict framework. In the rest of this section, we will discuss central issues in representing general chess rules and notions in various computer based frameworks. We will discuss three concrete examples – representation within the general purpose programming language C, within the constraint solver URSA and within the proof assistant Isabelle/HOL.[4]

3.1. General Chess Rules.

General chess notions can be strictly defined, for example, in terms of the structure of natural numbers, or within Zermelo-Fraenkel set theory (ZFC), or given axiomatically, via axioms in first order logic (FOL), or even using a general-purpose programming language (such as C or Haskell). For computer-supported reasoning about chess, the most reliable approach is to have explicit definitions of chess within a rigorous logical framework of some proof-assistant, which is usually some variant of higher-order logic (HOL) or type theory.

Chess is a complex game, with many rules, but it turns out that there are not many central notions that have to be used to state the properties like correctness of strategies for chess endgames that we are primarily concerned about. The chess game starts in an initial position, then two players play after each other and the game proceeds through a series of legal positions until one player wins or the game is drawn. Transitions between positions are made by legal moves played by the players. Therefore, to formally specify a chess game, one must represent *arbitrary chess positions*, the *initial position*, *legal positions*, *legal moves*, positions *won* for a player (its opponent is checkmated) and positions that are *drawn* (it is stalemate or a checkmate cannot be reached). Other specific definitions (e.g., the capture rules, or the promotion rules for a pawn) are just building blocks used to define the central notions.

We will follow Hurd[28] and make some important simplifications (since we are primarily focused on endgames). First, only pawnless games with no castling are considered. We also do not formalize three-fold repetition of positions, and fifty-move rule.[5] Also, the FIDE rules state that a position is legal if it is reachable from the initial position by a sequence of legal moves, but in the definition of legal positions (i.e., in the *lgl_pos* predicate) we omit this condition and the definition of the initial position. Namely, all positions legal in

[4]All formalizations, programs and proofs discussed in this paper are available online from `http://argo.matf.bg.ac.rs/downloads`.

[5]These two rules are not relevant for the strategy that we analyze: as it will be shown, it does not allow repetitions of positions, and it leads to a mate in 33 moves at most.

the strict FIDE sense will satisfy the conditions of our definition (there might exist some positions that satisfy our definition, but are not legal in the strict FIDE sense). So, since we show correctness of endgame strategies for all positions legal in a weaker sense, our proofs will be valid also wrt. the FIDE definition. Although a number of specific chess rules are missing and our theory is not a fully developed theory of the general chess, it still precisely defines notions relevant for pawnless endgames and gives us means to formally prove that our specific endgame definitions are in accordance with the general chess rules.

Instead of full details of a general chess formalization [28, 35], we will give only a rough outline. Since it needs to be strict and formal, we will present it in the style of the proof assistant Isabelle/HOL. The basic notions are the following.[6]

- The *side* is a datatype denoting two players (*White* and *Black*).
- Positions are represented by a type *pos*, characterized by the following components.
  - A function $on\_turn : pos \Rightarrow side$ gives the player that is on turn in the given position.
  - A function $on\_square : pos \Rightarrow square \Rightarrow (side \times piece)\ option$ gives the piece on the given square in the given position (or returning the special value $None$ if the square is empty). In the above, the datatype *piece* contains the chess pieces *King*, *Queen*, *Rook*, *Bishop*, *Knight* (note that we do not consider pawns) and the datatype *square* contains the squares on the board. The function implicitly ensures that there cannot be more than one piece on a square.

  The above notions are used in definitions of the following basic functions.
- The function $lgl\_pos : pos \Rightarrow bool$ checks if the given position is legal.
- The function $lgl\_move : pos \Rightarrow pos \Rightarrow bool$ checks if the second given position can be reached from the first given one by a legal chess move.
- The function $mate : pos \Rightarrow bool$ checks if the player on turn is checkmated in the given position.

The above notions for chess need to meet some conditions. For example, concrete definitions for *lgl_pos* and *lgl_move* have to ensure that legal moves can only be made between legal positions, so the following holds: $lgl\_move\ p_1\ p_2 \longrightarrow lgl\_pos\ p_1 \wedge lgl\_pos\ p_2$ (which we can prove as a lemma).[7] Also, after a legal move, the opponent is on turn, $lgl\_move\ p_1\ p_2 \longrightarrow on\_turn\ p_2 = opp\ (on\_turn\ p_1)$ must hold (where *opp* denotes the opponent side).

Some details of implementation in Isabelle/HOL. For illustration, we show some auxiliary definitions from our Isabelle/HOL formalization (that follows Hurd [28]) that are used in definitions of *lgl_pos*, *lgl_move*, and *mate*.

The *square* type is implemented as a pair of integers[8] – this enables to express many chess definitions succinctly, using arithmetic. *lgl_pos* definition must ensure that all pieces are within the board bounds, for which the function $on\_board\ (f, r) \longleftrightarrow 0 \leq f \wedge f < F \wedge 0 \leq r \wedge r < R$ is used (global constants $F = 8$ and $R = 8$, for files and ranks, determine the size of the board). We also define functions that for a given position p and a square sq check if

[6] Following the spirit of given representation, a general theory of two player strategic games (including chess) could be defined.

[7] Notice that such conditions hold for most (if not all) two player strategic games, so if we build a general theory of two player games, formulae like the given two would have a role of axioms.

[8] Instead of integers, natural numbers could have been used. However, integers allow expressing some properties using subtraction more easily.

sq is empty ($empty\ p\ sq \longleftrightarrow on\_square\ p\ sq = None$), or occupied by a piece of a given side *sd* ($occupies\ p\ sd\ sq \longleftrightarrow (\exists pc.\ on\_square\ p\ sq = Some\ (sd, pc))$).

Next we define scope of each piece. For instance:

$rook\_scope\ (f_1, r_1)\ (f_2, r_2) \longleftrightarrow (f_1 = f_2 \vee r_1 = r_2) \wedge (f_1 \neq f_2 \vee r_1 \neq r_2)$

In a position p, a square sq_1 attacks sq_2 if the *line between them is clear* ($clear\_line\ p\ sq_1\ sq_2 \longleftrightarrow (\forall sq.\ sq\_btw\ sq_1\ sq\ sq_2 \longrightarrow empty\ p\ sq)$)[9], and if there is a piece on sq_1 such that sq_2 is in its scope.

$attacks\ p\ sq_1\ sq_2 \longleftrightarrow clear\_line\ p\ sq_1\ sq_2 \wedge$
(*case* $on\_square\ p\ sq_1$ *of*
$None \Rightarrow$ False
$\mid Some\ (\_, King) \Rightarrow king\_scope\ sq_1\ sq_2$
$\mid Some\ (\_, Rook) \Rightarrow rook\_scope\ sq_1\ sq_2$ ...)

A side *sd* is *in check* in a position p if its king is on a square sq_1, and there is an opponent's piece on some square sq_2 such that it attacks the king on sq_1.

$in\_chk\ sd\ p \longleftrightarrow (\exists\ sq_1\ sq_2.\ on\_square\ p\ sq_1 = Some\ (sd, King) \wedge$
$occupies\ p\ (opp\ sd)\ sq_2 \wedge attacks\ p\ sq_2\ sq_1)$

Finally, a *position is legal* (denoted by the function *lgl_pos*) if the opponent of the player on turn is not in check, and, since we represent squares as pairs of integers, if all pieces are within the board bounds (coordinates of their squares are between 0 and 7, which is checked by the *on_board* function defined above).[10]

The notions of *legal move* and *checkmate* (i.e., the functions *lgl_move* and *mate*) are defined in a similar fashion. Since we will prove that our endgame strategy will always succeed in checkmating the opponent, we do not need to formalize the definition of a draw.

3.2. **Chess Endgames.** An alternative definition of the chess game can be given for a specific endgame, and the relevant chess rules can be described in simpler terms, leading to an *chess endgame definition*.

The type *pos* representing general chess positions can be significantly simplified if only the positions reachable during a specific endgame need to be considered (e.g., in a KRK endgame, only the rules for *King* and *Rook* are relevant and all other pieces and conditions describing their moves can be omitted). Simpler and more compact representations lead to more efficient reasoning.

Additionally, the functions on the type *pos* that formally describe the general chess notions (e.g., functions *lgl_pos*, *lgl_move* and *mate*) need not be executable (e.g., if definitions of those functions contain quantifiers, the computer framework need not be able to effectively compute if there is a checkmate in a given position). However, to aid some automatic reasoning approaches, it is desirable to have an executable representation at least for the endgame definition (e.g., it should be possible to compute effectively if a position is checkmate, if a move is legal, or to enumerate effectively all legal positions satisfying some given property).

[9]Since squares that a knight attacks are not on the same line with the square that it is on, the clear line condition is always satisfied.

[10]It may seem that the definition of legal position should require that the kings are not on adjacent squares. However, that condition is not necessary: if the two kings are on adjacent squares then, following Hurd's definitions, the player who is not on turn would be in check (his/her king would be attacked by the other king), which is not legal by the above definition.

In the ideal case, both simplified and general chess representation (and all corresponding definitions) should be described within the same computer framework (e.g., proof assistant). In that case, some morphism between the two representations can be defined, and the relationship between them can be formally shown. Further, all reasoning about the endgame properties should be done within the proof assistant leading to highest possible reliability. Reasoning about the endgame properties can also be done in some other systems (e.g., it can be done by using a constraint solver or some custom-designed C programs). In that case, the specific, simplified representation of endgame positions and rules have to be implemented in that system, but the relationship of such implementation with the general rules of chess can be shown only informally, leading to a lower degree of confidence.

In either case (a proof assistant, or some other system), the following notions can be used to represent the endgame rules.

- A type $[\![\texttt{EG}]\!]\texttt{.pos}$ represents chess positions encountered during a specific endgame.[11] It should be able to represent all chess positions that contain pieces relevant for the specific endgame (e.g., in the KRK case, two kings and the white rook) and all positions that can be reached from those during a play based on the strategy. Therefore, there can be a function $\widehat{\_} : [\![\texttt{EG}]\!]\texttt{.pos} \Rightarrow \mathit{pos}$ that is a bijection between $[\![\texttt{EG}]\!]\texttt{.pos}$ and this subset of *pos* i.e., a function such that each position $\texttt{p}$ from $[\![\texttt{EG}]\!]\texttt{.pos}$ represents some such position $p = \widehat{\texttt{p}}$ from *pos*. The endgame domain should be closed under legal moves in the general domain, i.e., $\forall\ \texttt{p}_1, p_2.\ \mathit{lgl\_move}\ \widehat{\texttt{p}_1}\ p_2 \longrightarrow (\exists\ \texttt{p}_2.\ p_2 = \widehat{\texttt{p}_2})$.
- The set $[\![\texttt{EG}]\!]\texttt{.inital\_positions} : [\![\texttt{EG}]\!]\texttt{.pos}\ \mathit{set}$ is the set of possible initial positions for the endgame, i.e., the set of all legal positions in $[\![\texttt{EG}]\!]\texttt{.pos}$ that contain all pieces relevant for the endgame (note that $[\![\texttt{EG}]\!]\texttt{.pos}$ can contain some positions where some pieces have been captured). It must hold that this set represents exactly all legal positions from *pos* that contain all relevant pieces (correctness of the endgame strategy is usually formulated for all plays that start in a position from this set).
- The function $[\![\texttt{EG}]\!]\texttt{.lgl\_pos} : [\![\texttt{EG}]\!]\texttt{.pos} \Rightarrow \mathit{bool}$ checks if the given position is legal. For any position $\texttt{p}$ of type $[\![\texttt{EG}]\!]\texttt{.pos}$ it must hold that $[\![\texttt{EG}]\!]\texttt{.lgl\_pos p} \longleftrightarrow \mathit{lgl\_pos}\ \widehat{\texttt{p}}$.
- The function $[\![\texttt{EG}]\!]\texttt{.lgl\_move} : [\![\texttt{EG}]\!]\texttt{.pos} \Rightarrow [\![\texttt{EG}]\!]\texttt{.pos} \Rightarrow \mathit{bool}$ checks if the second given position can be reached from the first given position by a legal move. It must hold that for any given positions $\texttt{p}_1$ and $\texttt{p}_2$ of type $[\![\texttt{EG}]\!]\texttt{.pos}$ it holds that $[\![\texttt{EG}]\!]\texttt{.lgl\_move}\ \texttt{p}_1\ \texttt{p}_2 \longleftrightarrow \mathit{lgl\_move}\ \widehat{\texttt{p}_1}\ \widehat{\texttt{p}_2}$.
- The function $[\![\texttt{EG}]\!]\texttt{.mate} : [\![\texttt{EG}]\!]\texttt{.pos} \Rightarrow \mathit{bool}$ checks if the player on turn is checkmated in a given position. It must hold that for any position $\texttt{p}$ of type $\texttt{KRK.pos}$ it holds that $[\![\texttt{EG}]\!]\texttt{.mate p} \longleftrightarrow \mathit{mate}\ \widehat{\texttt{p}}$.

The above conditions are necessary (and sufficient) to prove in order to show that if we prove some property for the endgame then that property holds wrt. the general chess rules, too.

3.2.1. *A Case Study of KRK.* We will present several instances of the KRK endgame definition. The relevant pieces are the two kings and the white rook (that could be captured and our representation needs to cover that situation, too).

[11] Note that we will use `typewriter` font for the endgame definition related notions (e.g., $[\![\texttt{EG}]\!]\texttt{.pos}$ or $\texttt{KRK.pos}$) and *normal italic* font for general chess notions (e.g., *pos*).

Isabelle/HOL and Records. The most natural way of representing the position is to pack all relevant information into a record (a structure) or an array. In Isabelle/HOL we define the following record.

```
record KRKPosition =
  WK   ::  "square" (* position of white king *)
  BK   ::  "square" (* position of black king *)
  WRopt   ::  "square option" (* position of white rook (None if captured) *)
  WhiteTurn   ::  "bool" (* Is white on turn?  *)
```

In order to represent a chessboard position, such record has to meet several conditions. First, the following condition checks whether all pieces are within the board bounds:

on_board (WK p) ∧ *on_board* (BK p) ∧ (¬WRcapt p ⟶ *on_board* (WR p))

WRcapt p denotes that the rook is captured in position p, i.e., that WRopt $p = None$. The following condition checks whether pieces are on different squares

WK p ≠ BK p ∧ (¬WRcapt p ⟶ WR p ≠ WK p ∧ WR p ≠ BK p)

Note that this definition uses the notion of *square*, and the *on_board* predicate which are also used in the definition of the general chess rules. Since both the general chess definition and the endgame definition are given in the same system (proof assistant), we could reuse such definitions.

KRK.pos is the type consisting of all KRKPosition records that satisfy the two conditions given above. The abstraction function $\widehat{\ }$ that maps KRK positions to general chess positions that they represent is defined as follows:

```
on_square p = λ sq. (if WK p = sq then Some (White, King)
                     else if BK p = sq then Some (Black, King)
                     else if WRopt p = Some sq then Some (White, Rook)
                     else None)"
```

Auxiliary functions that lead to the KRK.lgl_pos and KRK.lgl_move definitions are reformulated for the endgame. The following function checks if the black king is attacked, and is used in the definition of checkmate, along with the KRK.lgl_pos and KRK.BK_cannot_move definitions (that are not shown here, but are available in the formal proof documents).

KRK.WR_attacks_BK p ⟷
  ¬ WRcapt p ∧ *rook_scope* (WR p) (BK p) ∧ ¬ *sq_btw_hv* (WR p) (WK p) (BK p)

KRK.mate p ⟷ KRK.lgl_pos p ∧ KRK.BK_cannot_move p ∧ KRK.WR_attacks_BK p

The *rook_scope* definition is taken from the general chess formalization, but the notion of the black king being attacked is specific to KRK (*sq_btw_hv* call checks only if the white kings blocks the line between the rook and the black king horizontally or vertically) and such simple way is not correct wrt. the general chess rules (in general chess other pieces must be taken into account). Because of such differences, it is essential to have a formal link between the two layers.

C and Structures. In C, it is natural to represent squares by two integer coordinates. Also, there is no built-in option type in C so, for simplicity, we just add a flag that tells if the rook has been captured (if the rook is captured then its two coordinates become irrelevant). The type KRK.pos is the following:

```
typedef struct pos {
  bool bWhiteOnTurn;
  bool bRookCaptured;
  unsigned char WKx, WKy, WRx, WRy, BKx, BKy;
```

```
} KRKPosition;
```

For efficient storing, we represent each position by a bitvector of length 20: each of the three pieces by two triples of bits (as a triple of bits gives 8 possible values, corresponding to the default chessboard size) and two bits for representing which player is on turn and whether the rook has been captured. For instance, the position shown left in Figure 1, with white on turn, is represented by the numbers $(3,2)-(5,4)-(2,6)-1-0$, i.e,. by the following bitvector: 01101010110001011010. We implemented functions for transforming the above structure into bitvectors and back.

Unlike Isabelle/HOL, where the conditions that the record must satisfy are explicit, in C these conditions are ensured by additional functions.

It is easy to formulate all relevant chess notions and rules. For instance, the definition of mate is formulated in C as follows.[12]

```
bool Mate(KRKPosition p) {
    return LegalPositionBlackToMove(p) && BlackCannotMove(p) && WRAttacksBK(p);
}
```

URSA Constraint Solver, Bitvectors, and SAT. The URSA constraint solver is based on bitvectors and reduction ("bit-blasting") to SAT. In URSA, a position can be conveniently and naturally specified by six triplets of bits (two for each of the three pieces) plus bits for representing which player is on turn and whether the rook has been captured (as in the C version). Therefore, in URSA we represent positions with 20-bit numbers, and we developed procedures for transformation from individual pieces of information to 20-bit numbers and back.

Since the URSA language is C-like, it is easy to formulate all relevant chess notions and rules, very similarly as in the C version. For instance:

```
procedure Mate(nPos, bMate) {
  call LegalPositionBlackToMove(nPos, bLegalPositionBlackToMove);
  call BKCannotMove(nPos, bBKCannotMove);
  call WRAttacksBK(nPos, bBKAttacked);
  bMate = bLegalPositionBlackToMove && bBKCannotMove && bBKAttacked;
}
```

Once the URSA specification is made, by merits of the URSA system, we can immediately get a representation of properties of the KRK endgame in the language of SAT – bitvectors are vectors of Boolean variables and each URSA procedure call generates a Boolean formula constraining the parameters.

Linear Integer Arithmetic (LIA). Another possibility is to represent KRK positions and all relevant predicates using the language of linear integer arithmetic (LIA). The type `KRK.pos` then consists of integers $wk_{\mathtt{x}}$, $wk_{\mathtt{y}}$, $bk_{\mathtt{x}}$, $bk_{\mathtt{y}}$, $wr_{\mathtt{x}}$, and $wr_{\mathtt{y}}$, the Boolean `WhiteOnTurn`, and the Boolean `WRCaptured`. We constructed such specification in terms of LIA within the Isabelle/HOL proof assistant. Here is an example definition.

$\mathtt{LIA.rook\_scope}\ \mathtt{f}_1\ \mathtt{r}_1\ \mathtt{f}_2\ \mathtt{r}_2 \longleftrightarrow (\mathtt{f}_1 = \mathtt{f}_2\ \vee\ \mathtt{r}_1 = \mathtt{r}_2)\ \wedge\ (\mathtt{f}_1 \neq \mathtt{f}_2\ \vee\ \mathtt{r}_1 \neq \mathtt{r}_2)$

Note that these definitions are very similar to the ones based on records and the option type, but they use only LIA constructs. Because of that, the Isabelle/HOL system can

[12] Presented specifications in different frameworks are equivalent and substantially the same. However, there are still some minor differences (e.g., in naming conventions, in grouping of some conditions into predicates, etc), just as there are different programming styles.

automatically transform such definitions to the SMT-LIB input format and apply SMT solvers, which is the main method that we will use for reasoning in Isabelle/HOL.

3.3. **Chess Endgame Strategies.** Given the representation of the chess (endgame) rules, endgame strategies can be represented. Without loss of generality, we can only consider strategies for the white player. We can think about a strategy as a function that maps a given position to a position that is going to be reached after playing a strategy move. This function does not need to be total (w.r.t. the set of all positions with the relevant pieces on the board), for example, it need not be defined for positions that cannot be reached during the endgame. Instead of functions, sometimes a better choice could be to allow non-deterministic strategies and to model strategies by relations. Namely, non-deterministic strategies can be underspecified and can have much simpler definitions than corresponding deterministic functions – relations should describe only those aspects that are necessary for correctness, while aspects related to efficiency could be omitted from the specification and postponed (see Section 5.1). Therefore, we have the following choices for the strategy definition.

- A relation $WS_{rel} : pos \Rightarrow pos \Rightarrow bool$. There can be more than one position reachable by a strategy from a given position in one ply. The strategy must be *legal* i.e., it must give only legal moves:

$$WS_{rel}\ p_1\ p_2 \Rightarrow lgl\_move\ p_1\ p_2$$

- A deterministic-function $WS_{fun} : pos \Rightarrow pos\ opt$ (*opt* denotes the option type, like in Isabelle/HOL). This function corresponds to one specific instance of the strategy and returns the (single) move that white following the strategy should play in the given position (or the special value *None*, if the strategy is undefined for the given position).

If both the relation WS_{rel} and the function WS_{fun} are defined, then it is natural to require that they agree:

$$WS_{fun}\ p = p' \longrightarrow WS_{rel}\ p\ p'.$$

We also consider a non-deterministic function $WS_{set} : pos \Rightarrow pos\ set$, defined as $WS_{set}\ p = \{p' \mid WS_{rel}\ p\ p'\}$. Note that given the implementation of WS_{rel}, it can still be non-trivial to obtain an implementation of WS_{set}.

We will also consider the following relation:

- $B_{rel} : pos \Rightarrow pos \Rightarrow bool$ is a relation such that $B_{rel}\ p\ p'$ holds whenever a position p' can be reached from the position p by a legal move of black. Note that this is just a restriction of the relation *lgl_move* on the set of positions in which black is on turn.

Since the play of black is not restricted, we can't consider B_{fun} that would be analogous to WS_{fun}.

Although we assume strategies on the general chess position type *pos*, it suffices to define them only on the endgame type `KRK.pos` (e.g., we consider the relation $\texttt{KRK.WS}_{rel}$: `KRK.pos` $\Rightarrow$ `KRK.pos` $\Rightarrow$ *bool*). Every strategy definition on the type `KRK.pos` can naturally be lifted and yields a strategy on the type *pos*. Namely, given a strategy relation $\texttt{KRK.WS}_{rel}$ defined in the endgame terms, two positions p_1 and p_2 of the type *pos* are connected by the strategy relation WS_{rel} defined in the general chess terms, i.e., $WS_{rel}\ p_1\ p_2$ if and only if both can be represented by the type `KRK.pos` i.e., if there are positions $\texttt{p}_1$ and $\texttt{p}_2$ of the type `KRK.pos` such that $\widehat{\texttt{p}_1} = p_1$, $\widehat{\texttt{p}_2} = p_2$ and $\texttt{KRK.WS}_{rel}\ \texttt{p}_1\ \texttt{p}_2$ holds. If a strategy function is

lifted, then it returns *None* for all positions that cannot be represented by the type `KRK.pos`. Again, this is important for maintaining the link with the general chess game.

3.3.1. *A Case Study of the Strategy for KRK.* We have developed several implementations of the KRK endgame strategy described in Section 2.2.1, and then proved its correctness. Each implementation relies on some previously described KRK endgame representation.

Defining strategy conditions. For illustration, we show some auxiliary definitions that lead to the strategy definition. For example, in Isabelle/HOL the function that checks if a position $\mathtt{p}'$ can be reached from a position `p` by an *ImmediateMate* move is formalized as follows (assuming that auxiliary predicates `KRK.BK_cannot_move` and `KRK.WR_attacks_BK` were previously defined).

$\mathtt{KRK.immediate\_mate\_cond\ p\ p'} \longleftrightarrow \mathtt{KRK.BK\_cannot\_move\ p'} \wedge \mathtt{KRK.WR\_attacks\_BK\ p'}$

Similarly, the *RookHome* condition is formalized as follows (the divide attempt requires that the white rook is in a file or rank next to the white king):

$$\begin{array}{l}\mathtt{KRK.rook\_home\_cond\ p\ p'} \longleftrightarrow \\ \quad \mathtt{KRK.divide\_attempt\ p'} \wedge \\ \quad (\mathit{king\_scope}\ (\mathtt{BK\ p'})\ (\mathtt{WR\ p'}) \longrightarrow \mathit{king\_scope}\ (\mathtt{WK}\ p')\ (\mathtt{WR}\ p')) \wedge \\ \quad (\mathtt{KRK.BK\_cannot\_move\ p'} \longrightarrow \mathtt{KRK.WR\_attacks\_BK\ p'})\end{array}$$

The corresponding definition in URSA is very similar[13]

```
procedure RookHomeCond(nPos1, nPos2, bRookHomeCond) {
  call LegalMoveWR(nPos1, nPos2, bLegalMoveWR);
  call DivideAttempt(nPos1, nPos2, bDivideAttempt);
  call BKNextWR(nPos2, bBKNextWR);
  call WKNextWR(nPos2, bWKNextWR);
  call Stalemate(nPos2, bStalemate);
  bRookHomeCond = bLegalMoveWR && bDivideAttempt && (!bBKNextWR || bWKNextWR) && !bStalemate;
}
```

The corresponding definitions in C and LIA are also very similar.

Defining the strategy relation. The applied move must be the first one whose condition holds. Therefore, for each move we must have a function that checks if that move links the two given positions, and a function that checks if the strategy move is not applicable in a given position. Notice that these two are not just opposites of each other, since the latter requires rejecting all possibilities for this strategy move to be played from the given position. In Isabelle/HOL, we introduce the function `KRK.kings_move` $(f, r)\ k$ that for an index k between 1 and 8, gives coordinates of 8 squares that surround the given central square (f, r). Similarly, the function `KRK.rooks_move` $(f, r)\ k$ for k between 1 and 16 gives all squares that are in line with the rook (first horizontally, and then vertically). Combined, these give the enumeration of all possible moves of white (indices from 1 to 8 correspond to the king moves, and from 9 to 24 to the rook moves). We show this only for *ImmediateMate* in Isabelle/HOL, as other moves follow a similar pattern.

[13] The condition that the position is not stalemate makes one of small differences between the URSA and Isabelle/HOL specifications. In Isabelle/HOL, repeating the constraint on stalemate in conditions for all move kinds would give very large formulae, so in the definition of the strategy relation that condition is factored out and included only once, globally. On the other hand, URSA implements subformula sharing, so no significant overhead is incurred if a constraint is repeated several times.

Since we want to have all our definitions executable and we want to deal only with quantifier-free SMT formulae, we must introduce bounded quantification (that is unfolded into a finite conjunction). Then we can define predicates that encode that a certain kind of move cannot be applied.

$all\_n\ P\ n \longleftrightarrow \forall\ i.\quad 1 \leq i \wedge i \leq n \longrightarrow P\ i$

```
KRK.no_mate_WK p ⟷ all_n 8 (λ k.  let sq = KRK.kings_move (WK p) k in
   KRK.WK_can_move_to p sq ⟶ ¬ KRK.immediate_mate_cond p (KRK.move_WK p sq))
KRK.KRK.no_mate_WR p ⟷ all_n 16 (λ k.  let sq = KRK.rooks_move (WR p) k in
   KRK.WR_can_move_to p sq ⟶ ¬ KRK.immediate_mate_cond p (KRK.move_WR p sq))
KRK.no_immediate_mate p ⟷ KRK.no_mate_WK p ∧ KRK.KRK.no_mate_WR p
```

Note that the mating move can be performed only by the rook, and we have formally proved that in Isabelle/HOL, so the search for a mating move does not need to consider the moves of the king.

Since URSA, C and the quantifier-free fragment of LIA do not support quantifiers, conditions like the above are expressed by a finite conjunction or a loop.

Finally, we can introduce the relation `KRK.st_wht_move p p'` m, encoding that a position `p'` is reached from a position `p` after a strategy move of a kind m. We show only a fragment of the definition in Isabelle/HOL (the C, URSA and LIA definitions are very similar).

```
MoveKind = ImmediateMate | ReadyToMate | Squeeze | ApproachDiag |
   ApproachNonDiag | KeepRoomDiag | KeepRoomNonDiag | RookHome | RookSafe

KRK.st_wht_move p p' m ⟷
(if m = ImmediateMate then
   KRK.lgl_move_WR p p' ∧ KRK.immediate_mate_cond p p'
else
   KRK.no_immediate_mate p ∧
   if m = ReadyToMate then
      KRK.lgl_move_white p p' ∧ KRK.ready_to_mate_cond p p'
   else
      KRK.no_ready_to_mate p ∧
      ...
                     if m = RookSafe then
                        KRK.lgl_move_WR p p' ∧ KRK.rook_safe_cond p p'
                     else False)
```

Note that this is our strategy relation $\texttt{KRK.WS}_{rel}$, but it is parametrized by a move type (therefore, we can consider the relation $\texttt{KRK.WS}_{rel}{}^{\texttt{m}}$, where m is *ImmediateMate*, *ReadyToMate*, etc.).

Defining the strategy function. Although the strategy relation permits to play several different moves in some position, the ultimate goal is usually to reduce that to an executable function that calculates a single white player move for each position when it is on turn.

Defining deterministic strategy function $\texttt{KRK.WS}_{fun}$ requires a bit more effort. A function can iterate through all legal moves of white pieces until it finds a first move that satisfies the relational specification. An interesting exception is the *Squeeze* move. To make the strategy more efficient, the maximal *Squeeze* (the one that confines the black king the most) is always played (if there are several such moves, the first one found in the iterating process is used).

The strategy function definition in Isabelle/HOL uses several auxiliary functionals. The functional `KRK.first_move_WK p cond` takes a position `p` and a condition `cond` (a unary

predicate formulated on the set of all positions), iterates through all possible moves of the white king (using the function `KRK.kings_move`), and returns the index (a number between 1 and 8) of the first legal move (i.e., the move with the minimal index) that leads to a position that satisfies the given condition `cond`, or zero if there is no such move. The functionals `KRK.first_move_WR` and `KRK.first_move_white` are defined similarly. For example, the value of the expression `KRK.first_move_WR p KRK.immediate_mate_cond` is the index of the first mating move by the white rook starting from the position `p`, or zero if the black cannot be mated in one move. We also introduce the functional `KRK.min_move_WR p cond score`, that takes a position `p`, a condition `cond` and a function that assigns penalty scores to positions. The functional `KRK.min_move_WR` returns the index of the move of the white rook (a number between 1 and 16) that leads into the position that has the minimal penalty score among all such positions that satisfy the given condition `cond` (if there is more than one such move, the first one i.e., the minimal index is returned).

Using these auxiliary functions, the strategy function is defined in Isabelle/HOL as follows.

```
"KRK.st_wht_move_fun p =
   (let i = KRK.first_move_WR p KRK.immediate_mate_cond
     in if i > 0 then (KRK.move_white p (i+8), ImmediateMate)
   else let i = KRK.first_move_white p KRK.ready_to_mate_cond
         in if i > 0 then (KRK.move_white p i, ReadyToMate)
   else let i = KRK.min_move_WR p (KRK.squeeze_cond p) KRK.room
         in if i > 0 then (KRK.move_white p (i+8), Squeeze)
   ...
```

This definition always gives the *Squeeze* that maximally reduces the room.

This definition can be executed from within Isabelle/HOL (by means of `value` command) or its code can be exported in one of the supported functional languages (e.g., Haskell).

In URSA, the function is implemented similarly. A loop through all possible move indices is used to find the one that satisfies the current move condition.

4. Reasoning Methods

In this section we will describe several approaches for proving correctness of a given chess endgame strategy, i.e., for proving that the strategy for white is winning starting from any of relevant legal positions. We define that WS_{rel} is a *winning strategy* for white on a set of positions I with white on turn if all positions in I are *WS-winning positions* for white. A position is *WS*-winning for white if each play starting from it terminates in a position where black is checkmated, given that white follows the strategy.[14] More formally, *WS*-winning positions can be defined inductively: *(i)* A position is *WS*-winning, if white, following the strategy *WS*, immediately mates; *(ii)* A position is *WS*-winning if each strategy move by white followed by any legal move of black leads into a *WS*-winning position.

It is suitable, and sometimes even necessary to use computer support with chess rules and the strategy explicitly defined within a strict environment (proof assistant, theorem prover, constraint solving system, programming language etc.). Some of these proving approaches require that the implementation of the strategy is executable (i.e., that functions $\texttt{KRK.WS}_{rel}$, $\texttt{KRK.WS}_{set}$, or $\texttt{KRK.WS}_{fun}$ are implemented and used). We will assume that the

[14]Note that every WS-winning position is a winning position (assuming perfect play), but the opposite does not necessarily hold.

reasoning will be performed on the endgame level, while the link to the general chess rules should be ensured as discussed previously. Two approaches can be used.

Exhaustive retrograde analysis.: This approach assumes that the strategy is represented by a lookup table (endgame database) that assigns a strategy move to each relevant position. Then, using a retrograde procedure (in the style of Thompson's work [46]), it is verified that the endgame lookup table ensures win for white. If the strategy is represented algorithmically (e.g., by the function $\texttt{KRK.WS}_{fun}$), then the strategy move for each position is computed and stored into the lookup table. This approach is straightforward, but it does not provide a high-level, understandable and intuitive, argument on *why* the strategy really works.

High-level conjectures.: Within this approach, correctness of the strategy relies on several conjectures (e.g., invariants for various strategy moves, termination conditions) which, when glued together, imply that the strategy is winning. Conjectures can be proved either by enumerating all possibilities or by some more sophisticated reasoning methods, either manually or by using computer support. In the latter case, the conjectures can be proved either formally, within a proof assistant, or informally checked using a general purpose programming language or a constraint solver.

4.1. **Retrograde Analysis.** In this section we present a retrograde-style, enumeration-based procedure that can be used for showing correctness of a strategy[15].

Let I denote a set of all initial positions for which we claim that every play (with white following the strategy) starting from them will terminate with black checkmated. To apply the procedure, the set I must be closed under strategy moves of white followed by arbitrary legal moves of black, so every strategy play that starts from a position in I always remains in I. In the case of KRK we will assume that I equals the set `KRK.inital_positions` defined in Section 3.2 as a set of all legal positions with only the three relevant pieces on the board (for the strategy that we consider, the condition on I is met, which can be easily proved). Correctness of the strategy can be proved by showing that each position from I is a winning position.

WS-winning positions could be calculated by using a direct recursion, but it is much better to apply dynamic programming. For simplicity, we will assume that a deterministic strategy WS_{fun} is given, and defined for all positions p reachable from I and that functions *mate*, B_{set}, and WS_{fun} are executable. At the beginning of the procedure it is checked that I is closed in the above sense, i.e., that $B_{set}\ (WS_{fun}\ I) \subseteq I$. The set *Winning* will contain positions determined to be winning. It will be initialized to all positions from which white following the strategy immediately mates, and those positions will be removed from the set I. After that, the following is repeated. Among the remaining positions (which are not yet in the set *Winning*), the set S is found, containing all positions p such that: after a strategy move by white from p to p', it is not stalemate and all possible moves of black in p' lead to positions already in *Winning*. These are also the winning positions and we transfer them from the set I to the set *Winning*. The process terminates if all positions are determined to be winning (in that case, the set of remaining positions I is empty), or if there is no change made by the current iteration (in that case, the set S is empty). This procedure can be implemented within a function $retrograde : P\ set \Rightarrow bool$:

[15]A slightly modified algorithm can be used for computing look-up table for optimal play, as Thompson did [46].

```
function retrograde(I)
begin
   S := {p ∈ I | mate (WS_fun p)}
   Winning := S,   I := I \ S
   repeat
      S = {p ∈ I | B_set (WS_fun p) ≠ ∅ ∧ B_set (WS_fun p) ⊆ Winning}
      Winning := Winning ∪ S,   I := I \ S
   until I = ∅ ∨ S = ∅
   return I = ∅
end
```

The procedure runs in a BFS fashion and positions are added to the set *Winning* in increasing order of the number of moves needed to checkmate black. The central loop invariant is that after k iterations of the loop the set *Winning* contains all positions from I for which white that follows the strategy mates the black is mated after at most k moves of the black. From that, it can be easily proved that the strategy WS_{fun} is winning strategy on the set I iff the function *retrograde* returns `true`.

The procedure can also provide the longest possible game length, given that white follows the strategy.

Note that the analysis could be easily modified to use the non-deterministic definition of the strategy WS_{set} instead of the deterministic version WS_{fun}.

Note that although we have defined the function *retrograde* in the general chess terms, it can be implemented in chess-endgame terms.

4.1.1. *A Case Study of the Strategy for KRK.* Given the strategy implementation, it was rather straightforward to implement the above function in Isabelle/HOL and in C (the relevant part of the code was only around a hundred lines of code and took only half a day to write).

The retrograde analysis revealed some bugs in the initial implementation, and once they were fixed, confirmed that the strategy is correct. Therefore, this approach proved to be very suitable for rapid detection of bugs in the strategy implementation, without going into any deeper analysis of its properties.

There are 175 168 legal KRK positions with the three pieces on board and white on turn. It turns out that white always reaches win within 33 moves (i.e., within 65 plies). Due to the large number of positions and plies, this approach is hardly applicable without computer support. The check of correctness of the given KRK strategy using the C program is done in around 5s.[16]

4.2. **High-Level Conjectures.** Correctness of a strategy can be proved using a more abstract approach. The central statement can rely on a number of auxiliary, high-level conjectures (lemmas) that combined together lead to the correctness arguments (that the strategy WS_{rel}, i.e. its counterpart $[\![\texttt{EG}]\!].\texttt{WS}_{rel}$, is winning), but also provide insights into

[16]All running times are obtained on a cluster with 32 dual core 2GHz Intel Xeon processors with 2GB RAM per processor. All the tests were run sequentially, and no parallelism was employed. The URSA system was used with its default SAT solver – clasp (`http://www.cs.uni-potsdam.de/clasp/`). Although the solving process is deterministic, the running times can vary to some extent (no more than 10%), but since the exact information on time spent is not critical, we kept the experiment simple and performed all measurements only once.

why the strategy really works. Such auxiliary conjectures can be proved in different ways. In the following we will focus on the KRK strategy $\texttt{KRK.WS}_{rel}$, but the method can be easily applied to other strategies.

Many properties of the strategy can be formulated by lemmas of the following form ($\mathtt{p}_i$ are positions with white on turn, $\mathtt{p}'_i$ are positions with black on turn, and $\mathtt{m}_i$ are move types, e.g., *ImmediateMate*; a notation $\underset{i\in\{0..k\}}{\forall} \mathtt{p}_i\,\mathtt{m}_i\,\mathtt{p}'_i$ is just a shorthand for $\forall \mathtt{p}_0\,\mathtt{m}_0\,\mathtt{p}'_0\,\mathtt{p}_1\,\ldots\,\mathtt{p}_k\,\mathtt{m}_k\,\mathtt{p}'_k\,\mathtt{p}_{k+1}$):

$$\underset{i\in\{0..k\}}{\forall} \mathtt{p}_i\,\mathtt{m}_i\,\mathtt{p}'_i.\quad \mathtt{Pre}\;\mathtt{p}_0\;\wedge\;\underset{i\in\{0..k\}}{\mathtt{Seq}}\;\mathtt{p}_i\;\mathtt{m}_i\;\mathtt{p}'_i\;\mathtt{p}_{i+1}\longrightarrow\underset{i\in\{0..k\}}{\mathtt{Post}}\;\mathtt{p}_i\;\mathtt{m}_i\;\mathtt{p}'_i$$

The predicate `Pre` denotes preconditions for $\mathtt{p}_0$. For example, it could express that a position $\mathtt{p}_0$ is legal, or that all relevant pieces are on the board, but it could also give some additional constraints (for example, that the white king is closer to the white rook than the black king).

The predicate `Seq` denotes a sequence of strategy moves of white followed by legal moves of black. Again, $\underset{i\in\{0..k\}}{\mathtt{Seq}}\;\mathtt{p}_i\;\mathtt{m}_i\;\mathtt{p}'_i\;\mathtt{p}_{i+1}$ is just a shorthand notation:

$$\underset{i\in\{0..k\}}{\mathtt{Seq}}\;\mathtt{p}_i\;\mathtt{m}_i\;\mathtt{p}'_i\;\mathtt{p}_{i+1}\equiv\bigwedge_{i\in\{0..k\}}\left(\texttt{KRK.WS}_{rel}{}^{\mathtt{m}_i}\;\mathtt{p}_i\;\mathtt{p}'_i\;\wedge\;\mathtt{M}_i\;\mathtt{m}_i\;\wedge\;\texttt{KRK.B}_{rel}\;\mathtt{p}'_i\;\mathtt{p}_{i+1}\right)$$

Recall that the parameter `m` in the $\texttt{KRK.WS}_{rel}{}^{\mathtt{m}}$ relation denotes the type of the move played (e.g., $\texttt{KRK.WS}_{rel}{}^{\texttt{ImmediateMate}}\;\mathtt{p}\;\mathtt{p}'$ denotes that there is an immediate mate in position `p`, leading to the position $\mathtt{p}'$). Predicates $\mathtt{M}_i$ additionally constrain the strategy move types played by white (e.g., some $\mathtt{M}_i$ can require that $\mathtt{m}_i$ belongs or does not belong to some set of move types).

Finally, $\underset{i\in\{0..k\}}{\mathtt{Post}}\;\mathtt{p}_i\;\mathtt{m}_i\;\mathtt{p}'_i$ is just a shorthand for a postcondition that relates all positions and move types encountered during such a sequence of moves, i.e.,

$$\underset{i\in\{0..k\}}{\mathtt{Post}}\;\mathtt{p}_i\;\mathtt{m}_i\;\mathtt{p}'_i\equiv\mathtt{Post}\;\mathtt{p}_0\;\mathtt{m}_0\;\mathtt{p}'_0\;\mathtt{p}_1\;\ldots\;\mathtt{p}_k\;\mathtt{m}_k\;\mathtt{p}'_k\;\mathtt{p}_{k+1}$$

In most of useful statements, the postcondition is just a relation between the starting and the ending position i.e., it is of the form $\mathtt{Post}\;\mathtt{p}_0\;\mathtt{p}_{k+1}$.

Some variations of the above general form are allowed, and the series can end either by a black move (as given above), or by a white move.

Statements of the above form can express that some invariant is preserved or that some measure decreases after a series of moves (so it is a termination measure), or that some kinds of moves cannot or must be played after a series of moves, or that some series of moves leads to checkmate, and so on.

Note that we have assumed that lemmas are expressed in terms of the strategy relation $\texttt{KRK.WS}_{rel}{}^{\mathtt{m}_i}$. However, we could use the function $\texttt{KRK.WS}_{fun}{}^{\mathtt{m}_i}$ if it is available and we want to reason directly about it.

All central lemmas used in several available correctness proofs for Bratko's KRK endgame strategy [13, 32, 35] fit into the above form. For example, one of those lemmas claims that after a *basic move* (*Squeeze*, *Approach* or *KeepRoom*), only a basic or a *mate move* (*ReadyToMate*, *ImmediateMate*) can be played.

$$\forall \mathtt{p}_0\, \mathtt{p}_0'\, \mathtt{m}_0\, \mathtt{p}_1\, \mathtt{p}_1'\, \mathtt{m}_1. \quad \begin{array}{l} \neg \mathtt{WRcapt}\ \mathtt{p}_0\ \wedge \\ \mathtt{KRK.WS}_{rel}{}^{\mathtt{m}_0}\ \mathtt{p}_0\ \mathtt{p}_0' \wedge \mathtt{m}_0 \in \mathtt{KRK.BasicMoves} \wedge \mathtt{KRK.B}\ \mathtt{p}_0'\ \mathtt{p}_1 \wedge \\ \mathtt{KRK.WS}_{rel}{}^{\mathtt{m}_1}\ \mathtt{p}_1\ \mathtt{p}_1' \\ \longrightarrow \mathtt{m}_1 \in \mathtt{KRK.BasicMoves} \vee \mathtt{m}_1 \in \mathtt{KRK.MateMoves} \end{array}$$

Methods for proving lemmas. Although, in principle, statements of the above form can be proved manually (usually, only informally), we are interested in fully or semi-automated proofs within some computer system. Therefore, the statements have to be given in terms of the endgame definitions. Note that `Pre`, all $\mathtt{M}_i$, and `Post` should have an executable implementation.

- Since the set of positions `KRK.pos` and the set of available move types are finite, then this is a finite-domain conjecture that can be checked by an exhaustive enumeration. This approach can be more efficient if the functions $\mathtt{KRK.WS}_{set}$ and $\mathtt{KRK.B}_{set}$ defined in Section 3.3 are available so not all possible positions $\mathtt{p}_i$ must be considered, but only their subsets. Additionally, if the strategy is given by a function $\mathtt{KRK.WS}_{fun}$, then the search space (the quantification domain) is smaller (branching occurs only with the black moves). This can naturally be implemented in a language that enables iterating over sets of positions (so, it is simple in C or in Isabelle/HOL, but not in URSA).
- Such statements can also be considered as constraint solving problems and constraint solvers can be used to show that there are no positions and move types that would violate the implication, i.e., to show that

$$\mathtt{Pre}\ \mathtt{p}_0\ \wedge \bigwedge_{i\in\{0..k\}} \mathtt{Seq}\ \mathtt{p}_i\ \mathtt{m}_i\ \mathtt{p}_i'\ \mathtt{p}_{i+1} \wedge \neg \bigvee_{i\in\{0..k\}} \mathtt{Post}\ \mathtt{p}_i\ \mathtt{m}_i\ \mathtt{p}_i'$$

  is unsatisfiable. It can be formulated in a language suitable for a constraint solver, given the relations $\mathtt{KRK.WS}_{rel}$ and $\mathtt{KRK.B}_{rel}$, and the conditions `Pre`, $\mathtt{M}_i$ and `Post` (not necessarily effectively executable) can be formulated in the language of the constraint solver. In our case, the formula was either a bitvector-arithmetic (BVA) formula that was bit-blasted to SAT (via URSA) and solved by a SAT solver, or was a linear integer arithmetic (LIA) formula formulated within Isabelle/HOL that was converted to SMT representation (via Isabelle/HOL) and solved by an SMT solver.

Gluing lemmas together. Unlike lemmas, the central theorem often cannot be expressed in the language of a decidable theory. Namely, proving the central theorem may require, for instance, some inductive argument and using undecidable theories. In some computer-based proving approaches (e.g., in a general-purpose programming language or in some constraint systems) inductive arguments cannot be expressed, so the proof of the central statement must remain informal. On the other hand, proof assistants allow such forms of reasoning, so the central statement and its proof (using the lemmas) can be rigorously expressed and mechanically verified within the system.

4.2.1. *A Case Study of the Strategy for KRK.* In order to prove the central theorem, i.e., that the strategy for white is winning, we need to prove that it is terminating and partially correct. In this section we list all lemmas that together show that the strategy is correct (they are similar to lemmas given in [32], but somewhat improved, thanks to facts discovered while formally proving that they can be glued together).

Lemma 4.1. *After a strategy move by white, black cannot capture the white rook.*

Lemma 4.2. *After an ImmediateMate move, black is checkmated.*

Lemma 4.3. *ReadyToMate move leads to checkmate in the next move.*

Lemma 4.4. *RookHome and RookSafe are played only in the first three moves.*

Lemma 4.5. *Starting from a position with a room greater than 3, playing three full moves where white plays only basic strategy moves (Squeeze, Approach or KeepRoom) reduces the room or leaves the room the same, but decreases the Manhattan distance between the white king and the critical square.*

Lemma 4.6. *When the room is less or equal to 3, after a three full moves where white plays only basic strategy moves, the next move must be a mating move (ReadyToMate or ImmediateMate).*

All six lemmas can easily be formally stated in the general form described previously. For example, Lemma 4.3 can be formalized as follows:

$$\begin{aligned} \forall \mathtt{p}_0 \mathtt{p}_0' \mathtt{m}_0 \mathtt{p}_1 \mathtt{p}_1' \mathtt{m}_1 . \quad & \neg \mathtt{WRcapt}\ \ \mathtt{p}_0 \ \wedge \\ & \mathtt{KRK.WS}_{rel}{}^{\mathtt{m}_0}\ \mathtt{p}_0\ \mathtt{p}_0' \wedge \mathtt{m}_0 = \mathtt{ReadyToMate} \wedge \mathtt{KRK.B}\ \mathtt{p}_0'\ \mathtt{p}_1 \wedge \\ & \mathtt{KRK.WS}_{rel}{}^{\mathtt{m}_1}\ \mathtt{p}_1\ \mathtt{p}_1' \\ \Longrightarrow \quad & \mathtt{m}_1 = \mathtt{ImmediateMate} \wedge \mathtt{KRK.mate}\ \mathtt{p}_1' \end{aligned}$$

We formulated the same set of above lemmas in three different systems: C, URSA and Isabelle/HOL.

In Isabelle/HOL, encoding the lemmas is rather straightforward. For example, Lemma 4.3 is formulated as follows:

```
theorem ReadyToMateMove:
  "∀ p0 p1 p1' p2 t2. ¬ WRcapt  p0 ∧
                      KRK.st_wht_move p0 p1 ReadyToMate ∧
                      KRK.lgl_move_BK p1 p1' ∧
                      KRK.st_wht_move p1' p2 t2 ⟶
                        t2 = ImmediateMate ∧ KRK.mate p2"
```

Lemmas are formulated over the record-based representation, and are converted to the pure LIA and proved by applying SMT solvers. Translation is done manually, but could be automated by implementing a suitable tactic.

In URSA, the quantification is implicit and the statement is proved by showing that the negated statement is unsatisfiable:

```
call LegalPositionWhiteToMove(nPos1w, bLegalWhite1);
call IsRookCaptured(nPos1w, bRookIsCaptured1);
call StrategyRelation(nPos1w, nPos1b, bRel1, nsReadyToMate);
call LegalMoveBlack(nPos1b, nPos2w, bBlack1);
```

```
call StrategyRelation(nPos2w, nPos2b, bRel2, nStep2);
call Mate(nPos2b, bMate);
assert(bLegalWhite1 && !bRookIsCaptured1 && bRel1 && bBlack1 && bRel2 && !bMate);
```

In C, we used the brute-force enumeration-based approach to prove the lemmas. It required using nested loops that correspond to quantification (we won't further discuss this C-based approach).

Expressing the lemmas is simple in each of the above approaches. In the enumeration-based approach in C, proving lemma is actually execution of the code that expresses it. In Isabelle/HOL, the user must provide a proof, but that proof just needs to provide a boilerplate code that instructs the system to transform the lemma into the SMT-LIB form, run the external SMT-solver (we used the Z3 solver), import the answer and, possibly, verify it (in the case when proof-reconstruction is required and when the answer contains the proof-certificate). In URSA, proving lemma is automatically delegated to the underlying SAT solver by transforming the high-level specification into a propositional formula. The specifications using three approaches (C, URSA, and Isabelle) are rather short, even including formulations of lemmas – the C file has around 1000 lines, the URSA file around 2000 lines, and the Isabelle files around 7800. Table 1 (and Table 4 in a summarized form) shows statistics (the number of variables and clauses in the generated SAT instance and the running time) for verifying the specification that uses the deterministic strategy function (with no optimizations) using the constraint solver URSA.[17]

	Lemma 1	Lemma 2	Lemma 3	Lemma 4	Lemma 5	Lemma 6	Total
variables	38778	38935	77552	146180	117039	146198	564682
clauses	150911	151795	302173	575404	455909	575645	2211837
time	56s	6s	20s	6107s	2060s	36s	8285s

Table 1: Deterministic strategy function with no optimizations (the URSA approach)

5. Steps Beyond: Citius, Altius, Fortius

After proving correctness of the basic version of the strategy using different approaches, we want to make steps beyond, trying to perform proofs *faster*, to make them even *higher*-level, and to reformulate conjectures to be more general and *stronger*. We will go through a series of iterations using high-level statements and the constraint solver URSA. The culmination of that is a machine verifiable correctness proof (in Isabelle/HOL) for the strategy generalized to an $n \times n$ board, for arbitrary natural number n.[18]

5.1. Citius: Faster Computations and Efficiency Issues.

There are several ways to make checking and proving *faster*.

[17]The specification and the lemmas are slightly changed compared to the earlier version [32], thanks to the influence of combination of different proving approaches. The size of formulae and the running times are also slightly different.

[18]The strategy and the proof can be generalized to $m \times n$ case, but we will stick to the $n \times n$ case since it is more interesting as it admits using more symmetries in reasoning.

Using underspecifications. As discussed in Section 3.3, a non-deterministic strategy is underspecified, and its deterministic version has to give specific choices for each strategy move (there must be a way to choose if there are more than one move meeting the conditions). If the retrograde analysis is used, then it is faster to verify the deterministic strategy definition, than the non-deterministic one. However, the deterministic version becomes more difficult for analysis by high-level conjectures (as it is more complex). In addition, if correctness of the non-deterministic strategy has been previously shown, then it is sufficient to show that the deterministic version just refines its non-deterministic counterpart. This brings us to a more general and subtle idea, used often in interactive theorem proving (and actually used in our Isabelle/HOL and URSA proofs). We can reason about strategy moves, not only using their concrete definitions, but using only their preconditions and postconditions (that do not necessarily cover all details of the moves), as announced in Section 3.3. This way, our conjectures may be more general and much easier to verify (both in the constraint solving and in the interactive proving setting). While the executable function is capable of computing the single move that is played in each legal position and can effectively be used in a chess playing system, the strategy expressed in terms of a relation is much more general and covers various possible refinements.

For example, in the strategy in a form of a function, we used the *maximal Squeeze* move – one that maximally reduces the room for the black king. This can enable white to more quickly reach mate in some cases, but it affects the proving process. In the non-deterministic version, we use a loosely specified *Squeeze* (guaranteed to reduce the room, but not necessarily maximally).

The proving process is significantly more efficient when the relation is used, but the determinism is lost. If correctness of the deterministic function is still to be considered, then the total time should include time for the proof that the function meets the requirements of the strategy relation – 638 seconds. The total proving time is, therefore, 1382s (744s+638s) which is significantly smaller than 8285s, when only the function is considered (Table 4).

In the following, we will introduce some further optimizations, leading to new versions of both the non-deterministic and the deterministic strategy.

Using equivalent specifications. One of our main goals in developing formalizations of chess endgames (within different systems) is to have easily understandable, high-level descriptions of relevant notions and strategy moves, and all definitions used so far were designed with this goal in mind. However, they are often expressed in a way that, when unfolded, produces huge corresponding formulae. Consider, for instance, a formalization of mate in URSA:

```
procedure Mate(nPos, bMate) {
  call LegalPositionBlackToMove(nPos, bLegalPositionBlackToMove);
  call BKCannotMove(nPos, bBKCannotMove);
  call WRAttacksBK(nPos, bBKAttacked);
  bMate = bLegalPositionBlackToMove && bBKCannotMove && bBKAttacked;
}
```

The above description is elegant and easily understandable. However, when unfolded, it produces a huge corresponding formula. It can be described in a much more focused way. Namely, white can actually mate black only within a few patterns (for example, in the KRK endgame, the black king must be on one of the four edges). So, instead of the above general and readable definition, we can describe these patterns, as given in the following URSA specification (note that the condition that the position is legal can be dropped, since all mating positions are explicitly represented):

```
procedure MateOpt(nPos, bMate) {
  call Bitvector2Pos(nPos, nWKx, nWKy, nBKx, nBKy, nWRx, nWRy, bWhiteOnTurn);
  call AbsDiff(nBKy, nWRy, nBKynWRy); call AbsDiff(nWKy, nBKy, nWKynBKy);
  call AbsDiff(nBKx, nWRx, nBKxnWRx); call AbsDiff(nWKx, nBKx, nWKxnBKx);
  bMate = !bWhiteOnTurn &&
   (nBKx == 0 && nWRx == 0 && nWKx == 2 && nBKynWRy > 1 &&
    ((nBKy != 0 && nBKy != 7) || nWKynBKy <= 1) &&
    ((nBKy == 0 || nBKy == 7) || nWKy == nBKy)) ||
   (nBKy == 0 && nWRy == 0 && nWKy == 2 && nBKxnWRx > 1 &&
    ((nBKx != 0 && nBKx != 7) || nWKxnBKx <= 1) &&
    ((nBKx == 0 || nBKx == 7) || nWKx ==n BKx)) ||
   ...
}
```

The above specification generates a smaller formula, easier to digest by the solving process. However, there are still two major issues. First, while the former specification is simple, understandable and very likely without errors, the latter is complex and prone to errors. Here, the idea of *refinement* helps again: one first makes a specification in the style of the former one, then a specification in the style of the latter one, and then *checks* (using a constraint solver or a proof assistant) that these two are *equivalent*. Once this is done, in the further proving process, only the more efficient one can be used. Such process of refinement is often used in interactive theorem proving, but here we show that it is also applicable within constraint solving systems (and possibly in other proving approaches). The second issue is how the (optimized) specification can be derived in the first place. The answer is: by a careful analysis and again using computer support. One can start by a first incarnation of the optimized specification and check if it matches the high-level one. If it does not, then a computer system (for instance, URSA) provides instances where the two specifications do not match, the user fixes those differences and iterates the process. This is often demanding, but also rewarding at the end: one has simplicity, understandability, reliability, and efficiency.

The described approach was applied also to the stalemate definition, yielding additional speed-ups.

The refinement approach was used in simplifying the strategy in one more way: strategy moves specify what conditions should hold after the move, but not necessarily what piece white should move. For example, it can be proven that black can be mated only by the white rook, and not by the white king, and this fact can be used to simplify some definitions.

The alternative characterizations of mate and stalemate within URSA, gave a speed-up (Table 4). Note that the total time should also include the 0.3s needed to prove that two specifications for mate and that two specifications for stalemate are equivalent. It is not necessary to verify that the function meets the requirements of this newer version of the strategy relation – since the two relation-based specifications are proved to be equivalent.

Conditions that a certain move cannot be played in a given position, which are basic building blocks for the strategy relation definition (described in Section 3.3.1), can also be further refined and optimized (again carefully proving the equivalence with the original formulations). For example, the `KRK.no_immediate_mate` condition can be optimized by noticing that a mate move can be played only by moving the rook to an edge (where the black king is).

Table 4 shows the gain of using the optimized NoMove conditions (except for the `KRK.no_squeeze` condition that will be discussed in Section 5.3.1). Again, the total proving

time should include proofs that original definitions are equivalent to the optimized ones: under 5s altogether.

Exploiting symmetries. Chessboard has a number of symmetries and this can be exploited to (*i*) make the definitions simpler and more readable and (*ii*) to make reasoning more efficient. Symmetries in chess endings have already been studied, for example by Bain [2].

There are three basic symmetries – horizontal, vertical, and diagonal. Reflection functions map squares to squares: if F and R are global constants (denoting the numbers of files and ranks)[19] equal to 8, then the horizontal reflection $\mathcal{R}_h$ maps a square (x, y) to the square $(F - x - 1, y)$, the vertical reflection $\mathcal{R}_v$ maps it to $(x, R - y - 1)$ and the diagonal reflection $\mathcal{R}_d$ to (y, x) (the diagonal reflection is applicable only if the board is square). If a position is specified by the squares assigned to pieces on the board, then its reflected image is obtained by applying the reflection function to all those squares.

We define that a KRK position that has the black king on the square (bk_x, bk_y), the white king on the square (wk_x, wk_y), and the white rook on the square (wr_x, wr_y) is in *canonical form* if the triple $(2 \cdot bk_x + 1, 2 \cdot wk_x + 1, 2 \cdot wr_x + 1)$ is lexicographically smaller or equal to the triple (F, F, F), if the triple $(2 \cdot bk_y + 1, 2 \cdot wk_y + 1, 2 \cdot wr_y + 1)$ is lexicographically smaller or equal to the triple (R, R, R), and the triple (bk_x, wk_x, wr_x) is lexicographically smaller or equal to than the triple (bk_y, wk_y, wr_y) (if the rook has been captured, then the third components are ignored). Essentially, in canonical positions the black king is confined to be in a triangle that covers approximately one eighth of the board and the other two pieces are relevant only on the boards with odd dimensions, when the black king is on the central square or on the diagonal. If the black king is on the diagonal, then the white king must be on the diagonal or below it, and if it is also on the diagonal, then the rook must be on the diagonal or below it.

It is easy to define a function that checks if a position is in canonical form. Reflections can be used to map any position into canonical form and a canonization function (e.g., the procedure `Canonize` in URSA) can be easily defined as a composition of reflections. Canonization can be used to simplify definitions. For example, in URSA, the procedure `MateOpt` that we have previously shown can be simplified to the following one.

```
procedure MateOptSym(nPos, bMate) {
  call Canonize(nPos, nPosC);
  call Bitvector2Pos(nPosC, nWKx, nWKy, nBKx, nBKy, nWRx, nWRy, bWRCaptured, bWhiteOnTurn);
  call AbsDiff(nBKy,nWRy,nBKynWRy); call AbsDiff(nWKy,nBKy,nWKynBKy);
  call AbsDiff(nBKx,nWRx,nBKxnWRx); call AbsDiff(nWKx,nBKx,nWKxnBKx);
  bMate = !bWhiteOnTurn &&
      (nBKx==0 && nWRx==0 && nWKx==2 && nBKynWRy > 1 &&
           (nBKy!=0 || nWKynBKy <= 1) && (nBKy==0 || nWKy==nBKy)) ||
      (nBKy==0 && nWRy==0 && nWKy==2 && nBKxnWRx > 1 &&
           (nBKx!=0 || nWKxnBKx <= 1) && (nBKx==0 || nWKx==nBKx));
}
```

The stalemate definition in URSA is also very succinctly expressed using symmetries and canonization. Some other predicates can also be reformulated in such manner, yielding a more readable formalization. Table 4 shows the result of reformulating the definitions of mate, stalemate and *ReadyToMate* step. Note that this does not have a significant effect on the proving efficiency, but the main gain lies in readability and in conciseness

[19]Later we will also consider boards of other sizes than 8×8, so we consider symmetries in a bit more general framework.

of the specification. Again, the total time should also include 0.4s needed to prove that specifications are equivalent.

Symmetries and canonical positions can also significantly improve efficiency if they are used for the so called *without loss of generality (wlog) reasoning* [26]. The central lemma for exploiting symmetries (formally proved in Isabelle/HOL) states that if there is some property of chess positions invariant under all three kinds of reflections (if the property holds for a position, then the same property holds for its reflected image), then, in order to show that all positions satisfy that property, it suffices to show only that canonical positions satisfy that property.

```
theorem symmetry:
  fixes P ::  "KRKPosition ⇒ bool"
  assumes "∀ p.  P (reflectx_p p) ⟶ P p"
          "∀ p.  P (reflecty_p p) ⟶ P p"
          "∀ p.  P (reflectdiag_p p) ⟶ P p"
  assumes "∀ p.  is_canon p ⟶ P p"
  shows "∀ p.  P p"
```

We have proved in Isabelle/HOL that all relevant notions are invariant under all three types of reflections. For example, if the black king is checkmated in a given position, it is also checkmated in its reflected image (e.g. `KRK.mate` (reflectx_p p) $\longleftrightarrow$ `KRK.mate` p).

Since many notions are used, it is a tedious job to formulate all such lemmas in a proof-assistant, but once they are formulated, they are all almost trivial to prove (and almost all proofs can be obtained automatically). Therefore, any statement that has an outermost quantifier that universally quantifies over all positions can be relaxed by adding the condition that the position is canonical, which significantly reduces the search space.

Exploiting symmetries is formally justified in Isabelle/HOL (all described lemmas have been formally proved), while in URSA and C approaches it is used without justification within the system. Using the wlog symmetries in the URSA approach, led to additional significant speed-up shown in Table 4.

Since this is the final version in our chain of refinements, in Table 2 we present its detailed statistics.

	Lemma 1	Lemma 2	Lemma 3	Lemma 4	Lemma 5	Lemma 6	Total
variables	28190	27964	55958	111946	85186	111947	421191
clauses	98948	98166	196467	393049	299472	393060	1479162
time	3s	3s	6s	218s	72s	14s	316s

Table 2: Final version of the non-deterministic strategy

With all the presented optimizations, we built a new (final) version of the strategy function and again proved all the lemmas. The running times for proving lemmas are presented in Table 3, showing significant speed-up compared to the initial version of the deterministic strategy.

The presented optimizations affect the definition of the deterministic strategy as the optimized predicates are used both in the relation and the function definition. Since it was proved that the original function refines the basic strategy relation and that all reformulations were justified, the basic function also refines the final, optimized relation. Also, it can be separately proved that the optimized version of the function refines the optimized relation.

	Lemma 1	Lemma 2	Lemma 3	Lemma 4	Lemma 5	Lemma 6	Total
variables	33574	33718	67097	129518	101359	129536	494802
clauses	116391	116792	232535	450829	351951	450962	1719460
time	14s	3s	7s	2244s	1566s	15s	3849s

Table 3: Final version of deterministic strategy

This lemma (with 44668 variables and 154471 clauses) is proved in 559s, reducing the overall proving time for the function from 8285s, to 316s+559s=875s (plus 6s in total for proving the lemmas that justify the optimizations). This is still significantly less than 3849s needed for directly proving the lemmas for final deterministic version, which illustrates the power of the refinement used.

A summary of the effect of different optimizations in the URSA specification is given in Table 4.

	variables	clauses	time
Function (direct)	563872	2208153	8285s
Function (via relation)	626270	2448464	744+638s
Relation	394096	1480708	744s
optimizing Mate and Stalemate	380577	1374585	668s + 0s
optimizing NoMove	372987	1357965	569s + 5s
reformulations using symmetry	420987	1478400	524s + 0s
wlog reasoning	421191	1479162	316s
Optimized function (direct)	494802	1719460	3849s
Optimized function (via relation)	465859	1633633	316s + 559s

Table 4: Summary of the proving process for different version of URSA specification

Automation and efficient solvers/theories. Communication with external SAT/SMT solvers significantly increases automation in Isabelle/HOL. However, this requires formulating conjectures in appropriate theories. As we already noted, we were able to formulate all central lemmas in the language of linear arithmetic and this enabled their efficient, automated proofs using the Z3 solver integrated with Isabelle/HOL. This required to reformulate the definition of room to avoid multiplication, as discussed in Section 2.2.1. Otherwise, the theory of bit-vector arithmetic would have to be used, leading to less efficient proofs.

5.2. **Altius: High-level Proofs and Understandability Issues.** The understandability of the correctness proofs comes from the formulation of the central high-level lemmas. For example, in our current proof, there is a lemma that claims that *RookHome* and *RookSafe* moves can be played only in the first three moves (Lemma 4.4). We have demonstrated that this lemma can be formally expressed and proved fully automatically (for example, by using SMT solvers). However, this lemma can be replaced by three simpler ones. The first one claims that the *RookSafe* can be played only as a first move (the *RookSafe* cannot be played immediately after any strategy move of white, followed by a legal move of the black king). The second one claims that *RookHome* can be played only immediately after *RookHome* or

RookSafe. The third one claims that *RookHome* cannot be played immediately after two *RookHome* moves. These three lemmas together imply our original lemma, but also give us a higher-level understanding and more insight into the strategy details. Additionally, it turns out that it is much faster to prove three simpler lemmas: in URSA, the original lemma is proved in around 218s in the fastest variant, while the three simpler lemmas together require only around 58s. One reason for that is that the original lemma is too coarse and requires reasoning about 7 plies at the same time, while the simpler lemmas require reasoning about only up to 5 plies at the same time.

Further insights can be obtained by formulating explicit moves' preconditions, postconditions and invariants. Namely, in all previous approaches it remains only implicit what relationship between the pieces has been established after the first three moves, and what has exactly happened after the first, after the second, and after the third move. Finding explicit characterizations (expressed only in terms of positions and not the strategy) is a complicated task, but it would make the proof more understandable and would significantly contribute to deeper understanding of the very finest details of the strategy.

For example, *RookSafe* can be played only when no other move can be played. This condition is complicated (it takes into account the definitions of all moves in our strategy). However, by inspection and analysis of the positions in which this move can be played, we managed to characterize those positions explicitly by the following condition (shown in Isabelle/HOL).

$$\begin{aligned}&\texttt{(let } (\texttt{WKx}, \texttt{WKy}) = \texttt{WK p}; (\texttt{BKx}, \texttt{BKy}) = \texttt{BK p}; (\texttt{WRx}, \texttt{WRy}) = \texttt{WR p};\\ &\quad \texttt{C}_{\texttt{BR}} = \texttt{KRK.chebyshev\_dist (BK p) (WR p)}; \texttt{C}_{\texttt{BW}} = \texttt{KRK.chebyshev\_dist (BK p) (WK p)}\\ &\quad \texttt{in } (\texttt{C}_{\texttt{BR}} = 1 \wedge \texttt{C}_{\texttt{BW}} = 2 \wedge \neg \texttt{KRK.WR\_divides p} \wedge\\ &\qquad \texttt{WRx} \neq \texttt{WKx} \wedge \texttt{WRy} \neq \texttt{WKy} \wedge \texttt{WKx} \neq \texttt{BKx} \wedge \texttt{WKy} \neq \texttt{BKy}) \vee\\ &\qquad (\texttt{BKx} = 0 \wedge \texttt{BKx} = 0 \wedge \texttt{WKx} = 0 \wedge \texttt{WKy} = 2 \wedge \texttt{WRx} < 2 \wedge \texttt{WRy} > 2) \vee\\ &\qquad (\texttt{KRK.room p} = 2 \wedge \texttt{WKy} = 2 \wedge (\texttt{WKx} = 0 \vee \texttt{WKx} = 2)))\texttt{"}\end{aligned}$$

The first disjunct characterizes the positions where the white rook must escape towards a far edge as all other moves would leave it exposed, and the second and the third conditions characterize the positions where black must escape towards edge not to leave the black in a stalemate position (the third condition characterizes only two very special positions where the black king is confined to only a single square). A simple lemma is proved that claims that after any strategy move followed by a legal move of black the above condition for *RookSafe* cannot be satisfied, so *RookSafe* can be played only as a first move (when the postconditions of all moves are examined, it is almost obvious why this is so). This lemma ensures that *RookSafe* can be played only in the first move, but this time we have a rather explicit explanation and this proof could be done even manually.

Next, a lemma is proved that claims that after any two *RookHome* moves the following condition holds:[20]

$$\neg \texttt{KRK.WR\_exposed p} \wedge \texttt{KRK.WR\_divides p} \wedge \neg \texttt{KRK.in\_chk p} \wedge \texttt{KRK.room p} > 2$$

We follow Bratko's informal proof and prove that the following condition is preserved by all moves of black and all moves of white (except *ReadyToMate* and *ImmediateMate*):[21]

$$\neg\, \texttt{KRK.WR\_exposed p} \wedge (\texttt{KRK.WR\_divides p} \vee \texttt{KRK.L\_pattern p}) \wedge$$

[20]This condition is recognized by Bratko [13] (however, without the $\neg$ KRK.in_chk p condition that turns out to be necessary).

[21]In the white-to-move positions, the KRK.L_pattern p condition should actually be KRK.L_pattern′ p, where KRK.L_pattern′ is slightly changed condition KRK.L_pattern′ [13].

$$\neg \texttt{KRK.in\_chk p} \wedge \texttt{KRK.room p} > 2$$

The former condition ensures that a basic or a mating move can be played.

It turns out that lemmas that use such explicitly formulated pre and post conditions and invariants are much easier to prove than lemmas formulated only in terms of moves – all these lemmas with explicit invariants are together proved in under 10s in URSA which is a very significant speed-up compared to the 218s needed for the original lemma. Again, such explicit conditions bring not only speed but higher-level understandability.

To conclude, pen-and-paper proofs (e.g., the one given by Bratko [13]) required simpler lemmas, as complex lemmas are hard to prove manually. On the other hand, formulating just a few very coarse lemmas leads to a simpler proving process (although such lemmas can require higher proving time on the computer, they require less human time and effort to formulate them, which is the most significant and most time consuming component), so there is a trade off between the proof understandability and its efficiency.

5.3. **Fortius: Stronger Conjectures and Scalability Issues.** Having a conjecture and its proof at hand, we can consider if we can prove a *stronger*, more general conjecture. For instance, we can notice that the correctness of retrograde analysis is valid not only for chess, but for a wide class of games that can be defined as a loosely axiomatized theory. In this section we will focus on another sort of generalization — we will prove correctness of our KRK strategy for generalized, $n \times n$ chessboards.

5.3.1. *Generalization to $n \times n$ Chessboards.* Although the standard chess game is played on a 8×8 board, we can consider the KRK endgame and the presented strategy on $n \times n$ boards. This generalization, made in the spirit of mathematical generalizations, breaks the connection with the classic chess game, but illustrates power of the presented proving methodologies and also how they can be used for different games.

We easily modified our programs to be able to represent boards of other sizes than 8×8. Namely, bitvectors used for the board representation in URSA and in C can easily be adapted to a variable board size, by changing the number of bits used to represent coordinates. For example, in URSA, conversions from and to bitvectors can be modified to include the dimensions of the board.

Modifying the strategy. Rapid testing of this generalized version using the C program quickly revealed that the basic strategy does not define moves for all positions in the 4×4 and 5×5 cases. We made two adjustments to the strategy to cover those two cases.

First, in the *Approach* and *KeepRoom* moves of the original strategy it is required that the white king does not move to an edge in order to keep the king out of the edge where the black king is on, but, on small boards the white king might touch other edges. This does not make any problems, so we reformulated this condition and explicitly required that two kings are not on the same edge.

Second, the original strategy forbids making a *RookSafe* move such that the Chebyshev distance between the two kings is exactly two (unless they are both next to the white rook). This is needed to keep the black king from approaching the white rook in the next move, so the white rook would need to run to a safe position again. However, in several positions on the 4×4 and 5×5 boards it is not possible to make such *RookSafe* move (so it is not possible to make any move, since *RookSafe* is the last possible move of our strategy). Because of that, a new strategy move *RookSafeSmallBoards* (non-existent in the original Bratko's strategy)

had to be introduced at the end of the current strategy and it needs to be used (in certain positions) only in the 4×4 and 5×5 cases:

8. *RookSafeSmallBoards*: If none of the above is possible, then move the rook to an edge where the white king is (if not already on that edge); in the reached position, the Chebyshev distance between the white rook and the white king has to be 2.

The modified strategy is correct for all board sizes $n \geq 4$ (the additional move kind will be played only when $n = 4$ or $n = 5$).

We measured the total proving time needed for different chessboard dimensions and the results are summarized in Table 5. The C program used a retrograde analysis, and URSA was used to prove the lemmas about the correctness of the most optimized version of the relation.[22] Again, if correctness of the final strategy function is considered, then the total proving times by URSA for each n should include proofs that the function refines the final strategy relation. This additional time for $n = 4$ is 8s, for $n = 7$ exceeds the time used for proving lemmas – 316s, and for $n = 11$ exceeds our time limit – 1h.

n	No. of legal positions	plies to win	C	URSA
4	1312	21	0s	37s
8	175168	65	5s	316s
12	2360160	109	194s	1250s
16	14241920	153	847s	2904s

Table 5: CPU time (in seconds) required by the C approach and the URSA approach for proving strategy correctness for dimensions from $n = 4$ to $n = 16$

Reformulating the NoSqueeze condition. While the retrograde analysis proved its usefulness in preliminary experimenting (for example, in showing incompleteness of the generalized strategy for small boards), Table 5 shows that its running time grows quickly as n grows. The URSA and Isabelle approaches also becomes practically unusable for higher dimensions.

Meeting the limit in proving correctness for larger dimensions by any of the approaches needs new deep insights. For instance, a strategy move like *Squeeze* is played by the rook and, within the strategy description, all possible 14 moves by the rook are covered. For the 8×8 board, this approach does not do any harm. However, for the 1000×1000 case, the specification involves $999 + 999$ possible rook moves, this condition explodes and, together with other similar conditions, makes the conjecture impossible to resolve in a reasonable time.

If the maximal *Squeeze* cannot be played, then no *Squeeze* at all can be played. A deeper analysis reveals that any maximal *Squeeze* move fits into one of only 16 patterns, independent of the dimensions of the chessboard. Assume that the position of the white king, the white rook, and the black king after the *Squeeze* move are respectively (wk_x, wk_y), (wr_x, wr_y), (bk_x, bk_y). The rook must not be exposed after the move of white, so it must hold that $\max(|wk_x - wr_x|, |wk_y - wr_y|) \leq \max(|bk_x - wr_x|, |bk_y - wr_y|)$. The maximal *Squeeze*,

[22] It is interesting that the total size of formulae for the lemma and the time needed was greater for $n = 15$ than for $n = 16$. This is because of the representation used: for $n = 16$, four bits are used for coordinates and the conditions for ensuring that the coordinates are within the board disappear, contrary to, for instance, the case $n = 15$.

if it can be achieved, is played only in case of equality, i.e., if one of the following holds: $2wr_x = wk_x + bk_x$, $2wr_y = wk_y + bk_y$, $wr_x + wr_y = bk_x + wk_y$, $wr_x + wr_y = wk_x + bk_y$, $wr_x - wr_y = bk_x - wk_y$, $wr_x - wr_y = wk_x - bk_y$. Expressing wr_x and wr_y gives all candidate positions for the maximal *Squeeze*. If $2wr_x = wk_x + bk_x$, then $wr_x = (wk_x + bk_x)/2$, but if $wk_x + bk_x$ is odd, then $wr_x = (wk_x + bk_x + 1)/2$ is played. Also, if $wr_x + wr_y = bk_x + wk_y$, then either $wr_x = bk_x + wk_y - wr_{y_0}$ and $wr_y = wr_{y_0}$ or $wr_x = wr_{x0}$ and $wr_y = bk_x + wk_y - wr_{x0}$, where (wr_{x0}, wr_{y_0}) is the position of the white rook before *Squeeze*. Also, in some maximal *Squeeze* moves, the white rook moves to the file or rank next to one of $wr_x = bk_x + 1$, $wr_x = bk_x - 1$, $wr_y = bk_y + 1$, $wr_y = bk_y - 1$. This gives 16 candidate positions.

We used the above approach in URSA and Isabelle/HOL and we have shown equivalence between the optimized and the original definitions.

On the 8×8 board, the number of 16 potential positions for *Squeeze* is larger than the number of all possible positions for the rook (the rook can potentially move to one of the 14 squares). Therefore, we excluded this reformulation from our experiments for the 8×8 chessboard. However, as the dimension of the board rises, the number of possible moves by the rook increases, but the number of maximal *Squeeze* candidate positions remains the same. This is vitally important and enables us to efficiently reason about arbitrarily large boards, as it turns out that it is easy to characterize all move types with a number of possible candidate positions that does not depend on the board size (for *ImmediateMate* there are 4 candidate positions, for *ReadyToMate* there are 12, for *Squeeze* there are 16, *ApproachDiag*, *ApproachNonDiag*, *KeepRoomDiag*, and *KeepRoomNonDiag* are played by the king so there are 8 candidate positions for them, and for both *RookSafe* and *RookHome* there are only 4 candidate positions for the rook).

This reformulation not only makes a significant leap in scalability and efficiency of the proving, but it is also well-suited for communicating the strategy to human players (as they must consider a smaller number of candidate moves).

Making the board size arbitrary. The above approach made proving strategy correctness for large dimensions possible. But, still, these proofs are proofs for concrete, individual dimensions and not for arbitrary dimension. So, now we need another deep insight: all conditions used in specification of the strategy are expressed in terms of linear arithmetic, not only in terms of coordinates of the pieces, but also if the dimension is treated as a variable (and not as a constant)! This observation brings us to a general correctness theorem expressible in terms of linear arithmetic and provable by a proof assistant equipped with a support for SMT solving (for linear arithmetic). On the other hand, this technique is not applicable within the SAT-based URSA approach.

Therefore, when using the system equipped with support for reasoning in linear arithmetic, we can have a single theorem that shows that the strategy is correct for a chessboard of *any* size $n \times n$, for $n \geq 4$. The times to prove the lemmas for this theorem for the strategy relation in Isabelle/HOL are shown in Table 6.

We show the times for using Z3 in the oracle mode (without the proof reconstruction) and also in the fully verified mode.[23] Note that the proof reconstruction consumes most of the time (but also gives a strongest guarantees). The time for verifying the whole theory (everything except those six lemmas, including gluing them together) is around 259 seconds.

[23]The reported times are for Isabelle2015. The new SMT proof reconstruction module introduced in Isabelle2016 is significantly less efficient in our case than the older one, since it is tailored towards simpler and smaller proofs that occur in typical sledghammer tasks.

	Lemma 1	Lemma 2	Lemma 3	Lemma 4	Lemma 5	Lemma 6	Total
oracle	1s	1s	2s	76s	38s	18s	136s
verified	2s	2s	3s	803s	816s	576s	2202s

Table 6: Isabelle proofs for symbolic n

6. Related Work

We are not aware of another complex conjecture proved using three different computer-supported approaches. However, combining reasoning tools and approaches is present for decades and is used in a number of contexts and application areas. In the following text, we make just a brief selection of such works.

SAT solving has been used for solving very hard mathematical combinatorial problems like the Boolean Pythagorean triples problem [27]. There are integrations of FOL provers with SAT solvers [6]. SAT solvers can be used for solving CSP problems [45, 16, 43] and also SAT/SMT solvers can be combined with constraint programming solvers [44, 3]. Computer algebra systems have been plugged into SAT solver to provide a system that can be used as an assistant in proving process for either finding counterexamples or finitely verifying universal conjectures. This system has been used for proving a number of complex mathematical conjectures such as conjectures from graph theory regarding properties of hypercubes [50].

SAT solvers were used from the Isabelle system in proving the Erdös-Szekeres conjecture for convex polygons with at most 6 points [34]. SMT solver are interfaced to interactive theorem provers [8, 1, 39] and successfully used for complex tasks, such as verification of analog-mixed signal circuits [40]. FOL provers were used from the Isabelle system in formalizing Tarski's geometry [19].

A combination of constraint programming and theorem proving was used for software verification [18]. Combination of interactive and automated proving for FOL was used for reasoning about lazy functional programs [10]. Constraint programming is applied at the test suite reduction problem [24]. There are systems that combine testing and interactive theorem prover to reason about programs and to automatically generate concrete counterexamples [17].

Experimental mathematics is an approach to mathematics in which computation and experimentation are used to investigate mathematical objects and suggest conjectures, properties and patterns. Experimental mathematics is frequently based on general purpose computer algegra systems and on custom built software. There are research journals focused on this approach to mathematics, such as the journal *Experimental Mathematics*.

There is also a long history of computer based analysis of games and strategies. Computers have been often used for constructing chess endgame databases. Early programs for retrograde analysis were implemented by Thompson [46] and Bramer [11, 12]. Databases for a number of chess endgames are publicly available (e.g., the Lomonosov Endgame Tablebases, generated by Zakharov and Makhnichev, contain optimal play for all endgames with seven or less pieces).

Recently, computers have also been used to reason about endgame database correctness (e.g., to construct endgame databases that are correct by construction). Reasoning is based mainly on retrograde analysis and enumerations of positions and moves. Machine verifiable proofs of database correctness for chess KRK endgame database were given by Hurd [28, 29].

A combination of binary decision diagrams (BDDs) for representing positions, model checking tools for automation, and the proof-assistant HOL for high assurance were used.

Other games were also analyzed using computers. For example, Schaeffer et al. showed that checkers game (on an 8×8 board) is a draw [41]. Their argument included a giant endgame table, obtained by using massive computations combined with sophisticated search algorithms. Neither a high-level nor a machine verifiable proof was produced. Edelkamp applied BDDs to two-player games to improve memory consumption for reachability analysis and game-theoretical classification [21]. Gasser [23] showed that the game of Nine Men's Morris is a draw, using a combination of endgame databases and search.

Only a very few high-level endgame strategies are accompanied by correctness proofs. For instance, Zuidema [49] and Morales [36, 37] didn't prove correctness of their strategies for KRK. On the other hand, Bratko gave an informal, pen-and-paper proof for his KRK endgame strategy [13]. There are only few direct, computer-supported correctness proofs of strategies for chess endgames. A SAT-based constraint solver URSA was used by Maliković and Janičić to prove correctness of a KRK strategy closely related to the one considered in this paper [32]. Machine verifiable proofs were not provided and the lemmas cannot be glued together into a single theorem. A similar approach was used by Marić, Janičić and Maliković in Isabelle/HOL and automated SMT solvers were used to automatically prove the conjectures [35], leading to a self-contained, fully machine verifiable proof of the strategy correctness. We are not aware of other specifications of chess strategies within a proof assistant or a constraint programming system (the aforementioned work on verifying chess endgame databases [28, 29] does not deal with strategies understandable to humans).

7. Conclusions

In this paper we have shown how one can use computer support for proving correctness of a chess endgame strategy, and we advocate that the same methodology could be used for proving some other complex combinatorial conjectures (over finite domains). We have presented a case study of one particular problem and shown how one can reason about chess in a rigorous mathematical manner, supported by state-of-the-art computer tools. We have shown that KRK chess endgame is *strongly solved* [41], i.e. it is win for white for $n \times n$ chessboard, for each natural numbers n greater than 3. We revisit some key questions and list some of the main lessons that we learnt and that can be used for proving other combinatorial conjectures.

Specification language and verifiability.

- *Proofs developed within proof assistants have the highest degree of confidence.* Representing the general chess rules and the endgame in Isabelle/HOL leads to highest possible reliability.
- *Proof assistants have better expressive power than alternatives.* For instance, we could not glue the lemmas together (we could not even express the central theorem) in C and URSA, but we did do it in Isabelle/HOL.
- *Formalization should include an executable implementation of the analysed algorithm.* Defining an executable strategy function and relation enabled making various experiments (e.g., URSA made possible to find all positions that satisfy some criteria), but also directly enabled some reasoning methods (e.g, the retrograde analysis). Verified executable strategy formalization can be exported to a general purpose programming language.

Convincibility, faithfulness, and understandability.

- *Use a small set of basic definitions, separate basic and auxiliary definitions.* The basic definitions are cornerstones in any problem formalization and they must be concise, clear, and carefully manually inspected since the proofs are checked only modulo these. In our study, we invested a lot of effort in building a simple, understandable problem representation (in different settings) based only on general chess rules.
- *Introduce derived notions to speed up reasoning, but formally show connections with basic definitions.* We introduced many definitions specific for the KRK case, but we have always formally shown that they are in accordance with the general chess rules.
- *Introduce notions as abstractly as possible, so that they can be reused in other scenarios.* Identifying chess notions relevant for other two-player games can help analysing those games.
- *Use a problem representation understandable to humans, whenever possible.* Instead of a endgame tablebase, we focused on a strategy represented in a form of an algorithm, usable both by humans and computers.
- *Use symmetries as they can lead to more concise and understandable definitions, but also to more efficient reasoning.* Using symmetries should be justified by a meta-wlog theorem. We used symmetries for deriving optimized definitions of mate and other notions.
- *Clearly identify preconditions, postconditions and invariants.* The best understanding of algorithm comes by identifying the program state in all points of its execution, that can be described by lemmas that formulate preconditions, postconditions and invariants.
- *Optimized exhaustive search has both good and bad sides.* Retrograde analysis was a very efficient way to prove the strategy correctness both in C and Isabelle/HOL, but it did not provide us with better understanding, and proved to be inapplicable to large board sizes.

Abstraction, refinement, and generalization.

- *Use non-deterministic specification whenever possible and introduce deterministic specifications only when necessary.* We first introduced strategy in form of a relation, and defined the strategy function only in the end.
- *Reason about the most abstract definition that guarantees correctness and only afterwards introduce specific implementation details.* It was much faster to prove correctness of the strategy specification that leaves questions of choosing the squeeze move open.
- *Try to make definitions independent of the problem dimensions.* Finding only a finite number of candidate positions for each move enabled us to prove the strategy correct for arbitrary large boards.

Automation and efficiency.

- *Whenever possible, express relevant notions and conjectures in terms of theories supported by efficient automated theorem provers (for SAT, SMT, FOL, etc).* We adapted all notions so that they fit into QF-LIA (e.g., we had to change Bratko's notion of room, and replace quantifiers by finite conjunctions), which enabled us to heavily use *automation*: SMT solvers via Isabelle/HOL and SAT solvers via URSA. We relied on available automated procedures as much as possible, pushed them to their current limits and used human effort only for tasks requiring real insights and ideas.
- *Proofs by automated theorem provers have to be verified by proof assistants.* Connection between the URSA endgame definition and the exported SAT representation is not formally shown (and relies on the correctness of the URSA system implementation). On the other

hand, in Isabelle/HOL, transformation from the record-based representation to LIA can be implemented separately and LIA representation can then be automatically transformed into SMT-LIB (due to the SMT support in Isabelle/HOL).

- *If a candidate lemma cannot be proved by some computer tool, it is beneficial to get its counterexamples.* Counterexamples can give crucial hints on how to fix errors, and such corrections took majority of our time and effort. We used SMT solver within Isabelle/HOL that can give one counterexample (typically – a position that does not meet the statement), and – URSA that can list them all.
- *Use fast, even not maximally reliable tools for rapid testing of conjectures.* Checking of lemmas can be time-consuming and can take minutes. In contrast, the retrograde analysis can check the overall correctness of the strategy in only seconds, and therefore is much more suitable for rapid testing of variants of the strategy. However, the retrograde analysis does not provide the explanations of why the strategy works when it does and why the strategy does not work when it doesn't.
- *There is a trade-off between the coarseness of lemmas and time to prove them.* Coarse lemmas were easy to formulate, but required long time to prove. Splitting them to smaller lemmas required much more effort but brings pay off in much faster automated proofs.
- *Save time in proof evolution, not in the final formal proving.* The final, central theorem, within the Isabelle system, was proved in a few minutes. However, this time is not very important: once the theorem is polished, the proof is generated and verified only once. What is critical is that the road to this theorem requires checking and proving many conjectures and it is critical that these steps can be performed reasonably fast, possibly only in terms of seconds or minutes, so the proving process is really interactive. We believe that the presented synergy of different approaches enable this *seeking for the proof* approach practically usable in mathematical practice.

Our proofs are an illustration for a successful synergy among different computer-supported proving approaches. Our experience is that computer tools available nowadays provide a strong and reliable support for proving non-trivial conjectures from mathematics and computer science. This modern support for formal reasoning is a big step forward, analogous to a support for ground calculations that computers made available decades ago. Most important attributes of this modern support for reasoning are reliability and automation. Still, this support is far from being able to replace mathematician: it cannot provide intuition, deep insights, or proof ideas and there is no magic computer button for proving complex theorems. Rather, we show that one can use a mixture of proving approaches, methods, tools, ideas, and tricks as extremely valuable help in filling technical gaps in proofs and linking together a number of arguments.

Acknowledgement

The first and the second author are partly supported by the grant 174021 of the Ministry of Science of Serbia. The authors are grateful to the anonymous reviewers for their very detailed and helpful comments on this paper.

References

[1] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In *Proceedings of CPP 2011*, volume 7086 of *LNCS*. Springer, 2011.

[2] M. Bain. *Learning Logical Exceptions In Chess*. PhD thesis, University of Strathclyde, Glasgow, 1994.

[3] M. Banković. Extending SMT solvers with support for finite domain alldifferent constraint. *Constraints*, 21(4), 2016.

[4] H. Barendregt and E. Barendsen. Autarkic computations in formal proofs. *Journal of Automated Reasoning*, 28(3), 2002.

[5] H. Barendregt and F. Wiedijk. The challenge of computer mathematics. *Philosophical Transactions of the Royal Society*, 363(1835), 2005.

[6] A. Biere, I. Dragan, L. Kovács, and A. Voronkov. Experimenting with SAT solvers in Vampire. In *Proceedings of MICAI 2014*, volume 8856 of *LNCS*. Springer, 2014.

[7] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*. IOS Press, 2009.

[8] J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending sledgehammer with SMT solvers. *Journal of Automated Reasoning*, 51(1), 2013.

[9] S. Böhme and T. Weber. Fast LCF-style proof reconstruction for Z3. In *Proceedings of Interactive Theorem Proving*, volume 6172 of *LNCS*. Springer, 2010.

[10] A. Bove, P. Dybjer, and A. Sicard-Ramírez. Combining interactive and automatic reasoning in first order theories of functional programs. In *Proceedings of FOSSACS 2012*, volume 7213 of *LNCS*. Springer, 2012.

[11] M. Bramer. Correct and optimal strategies in game playing programs. *The Computer Journal*, 23(4), 1980.

[12] M. Bramer. Machine-aided refinement of correct strategies for the endgame in chess. *Advances in Computer Chess*, 3, 1982.

[13] I. Bratko. Proving correctness of strategies in the AL1 assertional language. *Information Processing Letters*, 7(5), 1978.

[14] I. Bratko. *PROLOG Programming for Artificial Intelligence (Third Edition)*. Addison-Wesley, 2001.

[15] I. Bratko, D. Kopec, and D. Michie. Pattern-based representation of chess end-game knowledge. *The Computer Journal*, 21(2), 1978.

[16] M. Cadoli, T. Mancini, and F. Patrizi. SAT as an effective solving technology for constraint problems. In *Proceedings of Foundations of Intelligent Systems*, volume 4203 of *LNCS*. Springer, 2006.

[17] H. R. Chamarthi, P. C. Dillinger, M. Kaufmann, and P. Manolios. Integrating testing and interactive theorem proving. In *Proceedings of ACL2 2011*, volume 70 of *EPTCS*, 2011.

[18] H. Collavizza, M. Rueher, and P. V. Hentenryck. CPBPV: a constraint-programming framework for bounded program verification. *Constraints*, 15(2), 2010.

[19] S. S. Djurdjević, J. Narboux, and P. Janičić. Automated generation of machine verifiable and readable proofs: A case study of Tarski's geometry. *Annals of Mathematics and Artificial Intelligence*, 74(3-4), 2015.

[20] B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for DPLL(T). In *Proceedings of CAV 18*, volume 4144 of *LNCS*. Springer, 2006.

[21] S. Edelkamp. Symbolic exploration in two-player games: Preliminary results. In *Proceedings of AI Planning and Scheduling, Workshop on Model Checking*, 2002.

[22] FIDE. The FIDE Handbook, chapter E.I. The Laws of Chess, 2004.

[23] R. Gasser. Solving Nine Men's Morris. *Computational Intelligence*, 12, 1996.

[24] A. Gotlieb, M. Carlsson, M. Liaaen, D. Marijan, and A. Petillon. Automated regression testing using constraint programming. In *Proceedings of AAAI 2016*. AAAI Press, 2016.

[25] T. Hales. Introduction to the Flyspeck Project. In *Mathematics, Algorithms, Proofs*, volume 05021 of *Dagstuhl Seminar Proceedings*. (IBFI), Schloss Dagstuhl, 2006.

[26] J. Harrison. Without loss of generality. In *Proceedings of Theorem Proving in Higher Order Logics*, volume 5674 of *LNCS*. Springer, 2009.

[27] M. J. H. Heule, O. Kullmann, and V. W. Marek. Solving very hard problems: Cube-and-conquer, a hybrid SAT solving method. In *Proceedings of IJCAI 2017*. ijcai.org, 2017.

[28] J. Hurd. Formal verification of chess endgame databases. In *Proceedings of Theorem Proving in Higher Order Logics: Emerging Trends*, number PRG-RR-05-02 in Oxford University CLR Report, 2005.

[29] J. Hurd and G. Haworth. Data assurance in opaque computations. In *Proceedings of Advances in Computer Games*, volume 6048 of *LNCS*. Springer, 2010.
[30] P. Janičić. URSA: A System for Uniform Reduction to SAT. *Logical Methods in Computer Science*, 8(3:30), 2012.
[31] C. Kaliszyk and J. Urban. Learning-assisted automated reasoning with Flyspeck. *Journal of Automated Reasoning*, 53(2), 2014.
[32] M. Maliković and P. Janičić. Proving correctness of a KRK chess endgame strategy by SAT-based constraint solving. *ICGA Journal*, 36(2), 2013.
[33] F. Marić. A survey of interactive theorem proving. *Zbornik radova, Matematički institut SANU, Beograd*, 18(26), 2015.
[34] F. Marić. Fast formal proof of the Erdös-Szekeres conjecture for convex polygons with at most 6 points. *Journal of Automated Reasoning*, 2017.
[35] F. Marić, P. Janičić, and M. Maliković. Proving correctness of a KRK chess endgame strategy by using Isabelle/HOL and Z3. In *Proceedings of Conference on Automated Deduction*, volume 9195 of *LNCS*. Springer, 2015.
[36] E. Morales. Learning patterns for playing strategies. *ICCA Journal*, 17(1), 1994.
[37] E. Morales. Learning playing strategies in chess. *Computational Intelligence*, 12(1), 1996.
[38] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
[39] Y. Peng and M. R. Greenstreet. Extending ACL2 with SMT solvers. In *Proceedings of Workshop on the ACL2 Theorem Prover and Its Applications*, volume 192 of *EPTCS*, 2015.
[40] Y. Peng and M. R. Greenstreet. Integrating SMT with theorem proving for analog/mixed-signal circuit verification. In *Proceedings of NASA Formal Methods*, volume 9058 of *LNCS*. Springer, 2015.
[41] J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen. Checkers is solved. *Science*, 317(5844), 2007.
[42] R. Seidel. Deriving correct pattern descriptions and rules for the KRK endgame by deductive methods. In *Advances in Computer Chess*, volume 4. Pergamon Press, 1986.
[43] M. Stojadinović and F. Marić. meSAT: multiple encodings of CSP to SAT. *Constraints*, 19(4), 2014.
[44] P. J. Stuckey. Lazy clause generation: Combining the power of SAT and CP (and mip?) solving. In *Proceedings of CPAIOR 2010*, volume 6140 of *LNCS*. Springer, 2010.
[45] N. Tamura, T. Tanjo, and M. Banbara. Solving constraint satisfaction problems with SAT technology. In *Proceedings of Functional and Logic Programming*, volume 6009 of *LNCS*. Springer, 2010.
[46] K. Thompson. Retrograde analysis of certain endgames. *ICCA Journal*, 9(3), 1986.
[47] M. Wenzel. Isabelle/Isar — a generic framework for human-readable proof documents. In *From Insight to Proof — Festschrift in Honour of Andrzej Trybulec, Studies in Logic, Grammar, and Rhetoric*, volume 10(23). University of Bialystok, 2007.
[48] F. Wiedijk, editor. *The Seventeen Provers of the World*, volume 3600 of *LNCS*. Springer, 2006.
[49] C. Zuidema. *Chess, how to Program the Exceptions?* Afdeling Informatica: IW. Stichting Mathematisch Centrum, 1974.
[50] E. Zulkoski, C. Bright, A. Heinle, I. S. Kotsireas, K. Czarnecki, and V. Ganesh. Combining SAT solvers with computer algebra systems to verify combinatorial conjectures. *Journal of Automated Reasoning*, 58(3), 2017.